FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Exploring the flexibility of Scala Implicits towards an Extensible Live Environment

Vasco André Costa Grilo

Master in Informatics and Computing Engineering

Supervisor: Hugo José Sereno Lopes Ferreira (PhD)

Second Supervisor: Tiago Boldt Sousa

February 13^{th}, 2013

Exploring the flexibility of Scala Implicits towards an Extensible Live Environment

Vasco André Costa Grilo

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Prof. Doutor Luís Filipe Pinto de Almeida Teixeira

External Examiner: Prof. Doutor Maximino Esteves Correia Bessa

Supervisor: Prof. Doutor Hugo José Sereno Lopes Ferreira

February 13^{th}, 2013

Abstract

This thesis presents the research and implementation details of an extensible live development environment for the Scala programming language called *Visual Scala*. It also describes the exploration of the flexibility of Scala's implicits towards such an environment. Scala is an Object-Functional programming language created by Martin Odersky which has been growing in popularity over the past few years.

When talking about development environments which support dynamic execution of code during runtime probably the most famous example are Read-Eval-Print Loops. REPLs are interactive top-level environments usually associated with dynamic or functional languages. Although they provide quick ways of inspecting and exploring a certain programming language, they stick to verbose textual representations of the intermediate values that are being generated by the programmer. We feel such an approach is rather limiting to our understanding of data structures and the effects of computation and so we propose to develop a development environment for Scala built on top of the Scala's standard REPLs with an associated library for converting type values to graphical representations.

Using the power of Scala's Implicits conversions we want to explore how flexible they can be when implementing an innovative mechanism of converting data structures to rich, interactive visualizations. As a web application, benefiting from HTML, CSS and JavaScript, along with the power of D3, a library for easily create graphical representations of data, Visual Scala offers an easily extensible and innovative conversions library for Scala's native type values. In this way, end users can interact with Visual Scala in a command-line fashion, with expressions resulting in rich graphical "widgets" instead of verbose text.

From reviewing the state of the art we find that there are similar tools which provide graphical representations of data structures, however they are usually intended for more technical and mathematical computing. The Julia Language is a very interesting example of such tools, although it provides graphical conversions in an explicit way and does not allow for new conversions to be created by the end-user. We feel that such a component, implicitly generating these graphics, for a more general purpose programming language, like Scala, is an interesting subject to explore and can have a significant impact on the community.

From implementing and exploring Visual Scala we conclude that Scala's Implicits provide high flexibility for developing a mechanism of invoking the right conversion for the right type value at hand. The resulting *Conversions Library* apart from being extremely flexible, is also easily extensible as it is not part of the running backend system and instead an external resource component. By evaluating user code by using the standard Scala's interpreter, Visual Scala also provides a "live" environment which maximizes graphical conversion of intermediate data throughout development.

Resumo

Esta tese descreve o estudo e os detalhes de implementação de um ambiente de desenvolvimento vivo e extensível para a linguagem de programação Scala, chamado de *Visual Scala*. Descreve também o processo de exploração da flexibilidade dos Implícitos de Scala quando se pretende caminhar em direcção a um ambiente desta natureza. Scala é uma linguagem de programação que mistura dois paradigmas, Orientado a Objectos e Funcional, crescendo em popularidade nos últimos anos.

Quando se fala em ambientes de desenvolvimento que suportam a execução dinâmica de código durante o seu *runtime*, o exemplo mais famoso são os *Read-Eval-Print Loops*. REPLs, ou Interpretadores, são ambientes interactivos de alto-nível, frequentemente associados a linguagens dinâmicas ou funcionais. Embora ofereçam um ambiente de rápida inspeção e exploração de uma linguagem de programação, restringem-se a apresentar os resultados e as estruturas de dados em formatos textuais bastante simples. Nós sentimos que esta aproximação é bastante limitativa na nossa percepção e análise das estruturas de dados e dos efeitos da computação sobre elas e como tal, propomos criar um ambiente para Scala que, construído com base no interpretador de Scala, oferece uma biblioteca de conversões gráficas de resultados.

Utilizando o poder das conversões implícitas de Scala, pretendemos explorar o quão flexível elas são quando se implementa um mecanismo inovador de converter tipos de dados para representações gráficas, ricas e interactivas. Apresentando-se como uma aplicação Web, beneficiando do HTML, CSS e JavaScript, juntamente com o poder de D3 (uma biblioteca de fácil geração de representações gráficas de dados), Visual Scala oferece uma inovadora e extensível biblioteca de conversões gráficas. Desta forma, os utilizadores podem interagir com o Visual Scala submetendo expressão a expressão, sendo que estas produzirão resultados gráficos em vez de apenas texto.

Da revisão do estado da arte descobrimos que já existem projectos similares, que oferecem representações gráficas de dados, no entanto, estas ferramentas estão geralmente associadas a contextos de computação tecnica ou matemática. The Julia Language é um exemplo bastante interessante de tal ambiente, no entanto, este tipo de ferramentas oferece representações gráficas de forma explícita e não permite a criação de novas conversões, por parte do utilizador. Nós acreditamos que esta componente de conversão implícita, dirigida a uma linguagem de programação de um âmbito mais geral, é um tema bastante interessante de ser explorado e pode criar algum impacto na comunidade.

Da implementação e exploração do Visual Scala, concluímos que os Implícitos de Scala permitem construir um mecanismo de invocar correctamente a conversão apropriada para o tipo de dados a converter de uma forma altamente flexível. A *Biblioteca de Conversões* resultante, para além de ser flexível, é também facilmente extensível pois ela não é parte integrante do sistema *backend* do Visual Scala. Ela faz parte do conjunto externo de recursos, em que modificações produzem efeitos imediatos na componente de conversão. Como a interpretação de código é feita recorrendo ao interpretador oficial de Scala, Visual Scala também oferece um ambiente "vivo" que maximiza a conversão gráfica de resultados intermédios enquanto se escreve código.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CSS	Cascading Stylesheets
CSV	Comma Separated Values
DOM	Document Object Model
FP	Functional Programming
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
I/O	Input/Output
JIT	Just-in-time
JSON	JavaScript Object Notation
LDE	Live Development Environment
OOP	Object-Oriented Programming
OFP	Object-Functional Programming
REPL	Read-Eval-Print Loop
SDE	Software Development Environment
SVG	Scalable Vector Graphics
UI	User Interface
VM	Virtual Machine
WWW	*World Wide Web*

Chapter 1

Introduction

The presented dissertation describes the process of exploring the flexibility of Scala's implicits towards an extensible live development environment called *Visual Scala*.

Nowadays, there is huge collection of software development tools available for programmers. Usually referred to as Software Development Environments, SDEs come in various forms and sizes. They range from simple libraries to complex development environments that can manage every stage of the software development life-cycle, from project source files management, source code editors to compilation, debugging, execution and even deployment tools. From simple interpreters to complex environments capable of coordinating large software development teams and manage the entire process of developing software from early user stories definition to implementation and deployment.

One example of a rather simple software development tool are Read-Eval-Print Loops (REPL). They are top-level interactive environments, usually associated with dynamic or functional programming languages, offering a quick way of evaluating expressions/instructions. REPLs follow a command-line approach in which end-users type one instruction or expression at a time and see them interpreted. Because of this interpreting approach, REPLs are usually referred to as *Interpreters*.

A rather significant element when developing software is output, produced from computation. Some code is meant for explicitly produce output, namely using Input/Output (I/O) modules of programming languages. When developing, programmers frequently use I/O code to print out data structures, current values of variables, and other relevant information. Output can be used to inform end-users relevant program's state information guiding its usage and workflow. On another level, as a popular way of debugging a program while in development phase, developers use I/O modules to output data structures to continuously check for desired outcomes of computation.

Interpreters, as they usually process user code one expression at a time, generate output for each input that is received. Typically this output is rendered in a verbose way, a textual representa-

tion of values produced by expressions[1]. While helpful, this verbose rendering offers little insight about the data structures that are being generated and modified.

In more mathematical-purposed programming languages, such as MATLAB for instance, it is useful to offer other representations of certain values than a verbose textual representation. If you think of a mathematical function like *cos(x)*, producing output like *f1: Function = cos(x)* offers no additional information from what the programmer already knows. Because of this, many tools for programming with these languages developed more visual, graphical representations of values. Such tools would render *2cos(x)* as a 2-dimensional graphical plot illustrating the behavior of the *cosine* function, among other helpful visual representations one could think of.

Output, and the interpretation that can be extracted from it, is very important and can help enriching and powering the end-user understanding of type values. If I'm developing some program that produces lists of numerals, those lists may, in different contexts, represent bar charts, statistical distributions, binary heap trees, etc. And so, a simple textual representation of the contained values in those lists is rather limiting to my quick understanding of the program's overall behavior.

When using such tools that already provide graphical representations of some type values, we find that tools often provide such visualizations in a explicit way. That is to say, end-users request such visualizations in an explicit way.

Graphical User Interfaces (GUIs) have become the major means of communicating and otherwise interacting with computer programs. Graphical visualizations leads to better understanding of data structures which provides, for computer science beginners, a stronger foundation and better preparation for their future. The human visual system and human visual information processing are optimized for multi-dimensional data [Mye86]. Clarisse [CC86] states that graphical programming processes information in a format close to the user's mental representations of them. And thus, can allow data to be processed in a closer way to how we process data in real world. Graphical representations of data or algorithms, have long been known to be helpful aids in program understanding [Smi77]. The work of Smith [Smi77] describes a number of psychological reasons for using visual displays for programs or data.

Motivated by this, we wanted to develop and explore a live, extensible development environment in which data structures have an implicit conversion to a visual representation. The biggest goal is to study how Scala's implicits can offer a flexible way of converting such data structures. Such environment, for end-users to program in the Scala programming language, aims to incorporate a top-level conversions library for Scala's type values that can be easily extensible as well as a liveness nature provided by evaluating code with the Scala's standard REPL, or interpreter. Scala expressions are evaluated with the Scala interpreter and then converted to an HTML version to be displayed on the web application.

1.1 Goals

The main goals of this dissertation are listed below:

[1] Expressions that don't perform I/O operations.

1. **Review dynamic development environments (i.e., REPLs)**

   What are these REPLs? Are they *actually* live environments?

2. **Explore the flexibility of Scala implicits towards an extensible live development environment.**

   How can we offer an implicit conversion library in a flexible way? Does it make it scalable? Extensible? Live? What can we gain from it? What will we possibly lose? What lessons can be learned from exploring such an environment?

3. *Secondary Goal:* **Research environments which offer graphical representations of data structures**

   How do these environments convert values to graphical outputs? Is this done in a explicit way or implicit way? Is there any related solution that already achieved our goals?

1.2 Methodology

A categorization proposed at Dagstuhl workshop [THP93], groups research methods in four general categories, quoted from Zelkowitz and Wallace [ZW98]:

- **Scientific method.** "Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis."
- **Engineering method.** "Engineers develop and test a solution to a hypothesis. Base upon the results of the test, they improve the solution until it requires no further improvement."
- **Empirical method.** "A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis."
- **Analytical method.** "A formal theory is developed, and results derived from that theory can be compared with empirical observations."

These categories apply to science in general however, effective experimentation in software engineering may require more specific approaches. Because sometimes software engineering research comprises computer science issues, human issues and organizational issues it is often convenient to use combinations of research approaches both from computer science and social science. The taxonomy described by Zelkowitz and Wallace [ZW98] identifies twelve different types of experimental approaches for software engineering, grouped into three broad categories:

Observational methods. "An observational method collects relevant data as a project develops. There is relatively little control over the development process other than through using the new technology that is being studied." There are four types: project monitoring, case study, assertion, and field study.

Historical methods. "A historical method collects data from projects that have already been completed. The data already exist; it is only necessary to analyze what has already been collected." There are four methods: literature search, legacy data, lessons learned, and static analysis.

Controlled methods. "A controlled method provides for multiple instances of an observational for statistical validity of the results. This method is the classical method of experimental design in other scientific disciplines." There are four types of controlled methods: replicated experiment, synthetic environment experiment, dynamic analysis, and simulation.

The optimal combination of methods to use in a concrete research approach is however, strongly dependent on the specific characteristics of the research study to perform, viz. its purpose, environment and resources. In this dissertation, the *Engineering Method* was used, in which we have established our hypothesis and developed a solution that would verify those hypothesis. Additional increments and re-development of the solution were made in order to improve the solution's validation of hypothesis.

1.3 Outline

This report is structured into three different parts, with the following contents:

Part 1: State of the Art The first part of this dissertation' report gives the reader an overview tour on Visual Scala's context and the motivation for the study of the flexibility of Scala's implicits towards the development of an extensible live environment for Scala. We'll describe the current state of the art on software development environments and development environments which offer graphical representations of data structures. We'll finalize by presenting a list and description of a few examples of such products that were found, as well an analysis on each product on certain desired functionalities for Visual Scala.

Part 2: Design and Implementation The second part contains our pragmatic approach to the problem at hand. Covering the design process, the methodology that was used as well as some design/technological decisions. Part 2 will continue on by extensively describing the implementation process, describring each of the system's components and how success was achieved.

Part 3: Conclusions The last part of this dissertation' report will present the conclusions from our collective work. Starting by summarizing the hypotheses and goals, the last chapter will cover the obtained results for every hypotheses. We'll continue on by describing the advantages and disadvantages of the system that was built and also describing the lessons we have learned from studying Scala's implicits flexibility towards the Conversions library. We'll finish this report presenting the future work to be done on Visual Scala.

Chapter 2

State of the Art

In this chapter, we'll cover the current state of the art regarding the context of our project. Beginning with a detailed explanation of the context in which Visual Scala emerges, we'll tour on different background concepts that are key to understanding some innovative traits of Visual Scala After we're done presenting concepts and related work on the field we'll finish this chapter by presenting some examples of related solutions that were researched. Each one of them will be described, presented with a screenshot of their usage and analyzed on some specific features we aim at Visual Scala so we can compare them to our project. The features that were topic of analysis can be read at section 2.2.

2.1 Context

The software development world is demanding and the speed and quality of how software is developed is critical to a project's success. Nowadays, little software is produced without the aid of software development tools. Tools can improve productivity and how fast you can develop code. Tools can also influence how well you learn a programming language and how well do you understand data structures.

Creativity thrives best if languages get out of your way and development environments submerge you in *a sea of live objects* [Ung07].

A survey by Tiobe Software, a software-quality consultancy that maintains an index of programming languages' popularity, says C, C++ and Java are the most popular. However, it also indicates that dynamically typed languages' adoption has increased recently [Pau07]. By comparing the quality of development environments, or software tools, from those available for statically typed languages and those available for dynamically typed languages, it's no wonder the former group has a higher adoption rate.

2.1.1 Type Systems

In computation, type systems can be classified as *trackable syntactic frameworks for classifying phrases according to the kinds of values they compute* [Pie02]. They are used with a programming

language to help reduce bugs in computer programs [CDJ+97].

Type systems provide conceptual tools to correctly judge adequacy of certain important aspects of language definitions. For instance, informal or natural languages often fail to specify type structures in enough detail in order to allow unambiguous implementations.

By extending the syntax of the language to include some sort of constraints to indicate *types* that can be associated to a certain value in the language, type checks can be performed to ensure consistency of programs when running. Program variables can assume different ranges of values during the execution of the program. As an example, a variable of type *Int* is assumed to only store values of that type throughout the entire execution life-cycle.

As part of type systems, type checking can happen at different program life-cycle phases such as compile time or run time, and even combinations of those two.

Programming languages where type checking happens at compile time are called *statically typed* or *static* languages. On the other hand, languages where type checking occurs at run time are called *dynamically typed* or *dynamic* languages. The latter group may not have types or have universal types that contain all possible values [CDJ+97].

These differences on programming languages nature shape the way and possible features certain software development tools can offer to end-users.

2.1.2 Static vs Dynamic Languages

As stated before, static languages enforce type checking at compile time. Because of this, the whole state of a program can be checked for consistency prior to execution. This translates in higher amounts of time for a developer to write his code since he has to worry about type consistency. However, this offers a great deal of advantages when building software tools to help development because continuous compilation of the code can provide type consistency errors, rich code completion or method suggestion regarding a variable type, etc.

On the other hand, as dynamic languages don't enforce these constraints, they use late binding of types and do them at run time. Due to the lack of type restrictions, this may translate in less worries for the programmer when coding and may result in lower amounts spent on type consistency and more time spent on creativity. In terms of performance however, dynamic languages are considerately slower in execution due to high amounts of type checking when executing a program.

Statically typed languages end up with large amounts of code lines as opposed to dynamic languages. Lower amounts of code is very appealing to programmers, procedures with several lines of code can sometimes be reduced to a single line. And while it may be appealing to any developer, when building large software systems, the possibility of having a huge number of code lines of dynamic languages can be maintenance nightmares [Yeg08] because they don't have static types.

From the mainstream's perspective, dynamically typed languages have finally come of age [Tra09]. They are used to build widely used real-world systems, often for the web, but increasingly

for domains that were previously the sole preserve of statically typed languages (e.g. [KG07]), often because of lower development costs and increased flexibility [MM90, Nor92].

2.1.3 Software Development Environments

There are different taxonomies for tools that help programmers develop code. These tools can range from small tool-sets used by a single developer to full-on, complex environments that helps managing and coordinate large software development teams.

Software Development Environments (SDE) are large sets of tools that support the coding process as well as any other related software development activities. They usually offer a GUI-based environment with all sorts of functionalities from creating projects, to editing source code files, managing library dependencies, compiling and running the code as well as debugging it and sometimes deploying it.

2.1.3.1 Integrated Development Environments

Integrated Development Environments (IDE) can be classified as SDEs. An IDE is software that helps programmers with many stages of software development process. IDEs contain all functionalities of those described for SDEs and in some cases, they are even richer, containing interpreters, version control systems, GUI construction tools, etc. They provide a helpful abstraction regarding the needed configuration to piece together all steps into a cohesive unit and as complex systems, there is more than one application running simultaneously that are perceived by the end-user as just one single program.

IDEs for statically typed languages are very mature and very capable of increasing productivity [Jav10]. Usually an IDE is bound to a specific programming language, matching the language paradigm to the available feature set, although there are multi-language IDEs.

Type checking at compilation time paves way to a huge set of useful information that can be gathered while the programmer is typing his code. Everything that is to know about a certain variable like related methods or applicable operations can be detected continuously and result in rich code suggestion/completion to the end-user, among other features. Rich refactoring, specially in a project-wide range can be provided as well when using static languages. As such, IDEs for statically typed languages have been evolving since their dawn and they almost offer everything a programmer needs to efficiently improve software development.

Sadly, in comparison, software development tools for dynamic languages do not show the same maturity and do not offer as much features as those available for static languages. Due to not enforcing type consistency at compilation time, information about variable types can only be obtained when running the program and inspecting it's state. This results in little information available when the programmer is typing his code.

So, what features does one IDE for a dynamic language should have? Ted Leung [Leu08] describes such tools with the need to support most refactoring described in the book *"Refactoring:*

Improving the Design of Existing Code" [Fow99] while still maintaining performance and a small footprint. Bernstein wrote on the relevance of dynamic debuggers for dynamic languages [Ber10].

The gap on the quality IDEs offer for developing with static and dynamic languages is wide due to, we believe, applying the same design concepts that were created to solve problems in a static context. Common IDE features like static code analysis in dynamically typed languages are possible but are unable to completely solve programmer's basic needs such as project-wide refactoring, auto-complete, advanced debugging techniques or advanced error detection. Understanding dynamic types [ACPP89] is very hard, if not impossible, to analyze through static parsing.

Is it possible, to bring the dynamism from dynamic languages to statically typed languages?

2.1.3.2 Live Development Environments

Live Development Environment (LDE) concepts are not, per se, a new idea. They have been around since the early days of reflective languages.

In 1996 Squeak, an IDE for the programming language Smalltalk, offered a fully reflective environment with real live objects, where the IDE is part of the runtime environment of the applications it hosts [Squ12]. This approach and principles offered a new reality for programming with Smalltalk. Smalltalk is a reflective, object-oriented, dynamically typed programming language that originated in 1972.

Squeak was built on top of a Smalltalk Virtual Machine (VM) and provides a fully portable, self-hosted development environment for the language. It can be used for a wide range of projects from multimedia applications, educational platforms to commercial web application development. This IDE provided a live system, so there is no need to lengthy recompilation and restart. Anything you modify will be compiled and take effect immediately.

Implementing the development environment in a reflective and dynamic language allows to leverage the runtime infra-structured to assist on typical IDE functionalities, and gives the programmer an environment where creating, modifying and running code are no longer separated activities. Bracha claims that with a dynamic IDE, or a somewhat LDE, we can have live and inspectable code, opposed to dead compiled code from static contexts [Bra08], hence the name *Live* Development Environment.

Squeak provides the best example of what a live development environment should be. But, why do current software development tools for programming with dynamic languages still lack maturity when comparing them to those for static languages?

We believe that such a gap comes from applying principles that work on static contexts, in statically typed languages, to dynamic contexts from dynamic languages. For example, in a static context, an IDE can easily inspect (without actually running) the program for structural components, such as classes, methods, fields, and thus it can acquire a complete model from it. This is paramount to provide accurate code-completion functionalities as well as refactoring facilities.

Dynamic languages do not share the same rationale. Coupled with an extensive native usage of meta-programming, as to say, manipulating structural and behavioral elements during runtime,

IDEs for dynamic languages have been uncommon, and surely not on par with respect to modern expectations.

Without executing the program from a dynamic language code, it would be undecidable to list, for example, the methods available for the programmer to invoke in the context of a certain class. As to overcome these problems, the LDE would have to gain knowledge of the executing environment, thus becoming an integral part of the runtime infrastructure. The program would have to be, partially, executed while it was being written.

2.1.4 Read-Eval-Print Loop

As a type of *live environments*, Read-Eval-Print Loops (REPL) are known as an interactive top-level environments. They are simple and interactive. Usually the term is used to refer to a Lisp interactive environment, however it can be applied to command line shells or similar environments.

The name comes from the names of the original Lisp primitive functions which enable this functionality, namely:

1. **read** responsible for accepting expressions from the user and parse it into some data structure.

2. **eval** responsible for evaluating the internal data structure, result of the read function.

3. **print** responsible for printing out the result of the eval function. If complex, it may be pretty-printed making it easier to understand.

Providing quick feedback to the user, REPLs often come as an essential part of learning a new programming language.

REPLs can be seen as live environments, where the user continuously submits expressions to be evaluated and the system's state is iteratively evolving with no need for re-evaluating previous expressions. They don't, however, allow for insertion of than one expression in a continuously evaluation process. It is possible to input several expressions at once but they will only be evaluated when the programmer triggers the end of that code block.

One can argue if REPLs are truly live systems or rather a tight background evaluation cycle, however, from a programmer's coding perspective, they can be perceived as actual live environments. Powerful tools that they are, REPLs are not suitable environments for building large software systems with several source files and dependencies between them, as well as external libraries or components.

2.1.5 Reflection

Reflection can be shortly defined as the property of a system that allows to observe and alter its own structure or behavior during execution. Brian Smith in 1982 first introduced the notion of computational reflection in programming languages in his doctoral dissertation [Smi82] and we

can categorize reflective mechanisms in two major forms: *structural* and *behavioral* reflection [JF96].

Reflection is something that is normally achieved through usage and manipulation of (meta-)data representing the state of the program/system. There are two aspects of such manipulation: (i) *introspection*, i.e., to observe and reason about its own state, and (ii) *intercession*, i.e., to modify its own execution state (structure) or alter its own interpretation or meaning (semantics) [Caz98].

Languages like Lisp, Prolog, Smalltalk, among others, have structural reflection mechanisms for a long time now, however, behavioral reflection has not been so clearly tackled yet, maybe because it touches aspects governing the semantics of programs. Malefant *et al.* stated that when confronted with behavioral reflection, most language implementors adopt interpretive techniques. Interpreters ease modifications and react to them as soon as they occur, thus a remarkable advantage in reflection [JF96].

2.1.6 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm in which concepts are represented as *objects*, that have data fields, i.e., attributes that describe the object, and associated procedures/behaviors commonly known as *methods*. Objects interact with one another for designing applications and computer programs [KK11, McC62]. Objects are instances of *Classes*. Classes are the static representation, or definition, of an object, and multiple objects can be created from the same class, in order to represent variations of that Class.

One can say, in the conventional model, that a program is seen as a set of subroutines, or tasks, to be performed. Opposed to this, in OOP, a program may be seen as a collection of objects, each one capable of receiving messages, processing data and sending messages to other objects. The collection and interaction between these objects define the program's overall workflow and behavior. Every object can be viewed as independent entities with distinct and clearly defined roles, or responsibilities. It is very common for OOP data structures to have their own subroutines associated to them, called methods, providing the behavior alongside the actual data, which is represented by the object's attributes. These methods act as the intermediaries for retrieving or modifying the data they control. In this sense, it is safe to say that the programming construct that combines data with a set of operations for their access and management, are called objects.

Mainstream object-oriented languages are class-based. Some examples comprise Simula, Smalltalk, C++, Modula-3 and Java [AC96].

Object-Oriented Programming has roots that can be traced back to 1960s, and Dr. Alan Kay, is considered the one who coined that term and claims that a detailed understanding of LISP internals was a strong influence on his original thinking of OOP [Kay03]. Citing Dr. Kay, this is what OOP means to him [Kay03]:

> *"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and*

in LISP. There are possibly other systems in which this is possible, but I'm not aware of them."

2.1.7 Scala

The name *Scala* stands for "scalable language" [OSV08], as it was designed to grow with the demands of its users. Scala runs on the standard Java platform and inter-operates seamlessly with all Java libraries.

We can safely say that, Scala is somewhat of a blend between object-oriented (OOP) and functional programming (FP) concepts in a statically typed language. The two programming styles have complementary strengths when it comes to scalability [OSV08]. Scala's functional programming constructs make it easy to build things quickly from simple parts and its object-oriented constructs make it easy to structure larger systems and to adapt them to new demands.

Scala was developed with the premise that you could mix together object orientation, functional programming, and a powerful type system and still keep elegant, succinct code [KS12]. The combination of both styles in Scala makes it possible to express new kinds of programming patterns and component abstractions. In Scala, functions are objects. Programs can be constructed through both the definition and composition of objects or functions. This gives Scala the ability to focus on "nouns" or "verbs" in a program, depending on what is the most prominent. Scala also blends expressive syntax with static typing.

Mainstream statically typed languages tend to suffer from verbose type annotations and boilerplate syntax. Scala takes a few lessons from the ML programming language [KS12] and offers static typing with a nice expressive syntax. Scala made choices about syntax that drastically reduced the verbosity of the language and enabled some powerful features to be elegantly expressed, such as type inference.

Taken from the official website's description of Scala [Scaa]:

Scala is object-oriented Scala is a pure object-oriented language in the sense that *every value is an object* [Scab]. Types and behavior of objects are described by classes and traits. Classes are extended by subclassing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

Scala is functional Scala is also a functional language in the sense that *every function is a value* [Scab]. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying. Scala's case classes and its built-in support for pattern matching model algebraic types used in many functional programming languages.

Scala is statically typed Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner. The type system supports generic classes, variance annotations, upper and lower type bounds, compound types, explicitly typed self

references, views and polymorphic methods. A local type inference mechanism takes care that the user is not required to annotate the program with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

2.2 Related Solutions

Following the overview tour on some background concepts and the context of this dissertation, we'll continue on by presenting researched related solutions. Several tools were searched and analyzed regarding some important traits when comparing them to our goals for Visual Scala. We analyzed each tool regarding the possibility of: *dynamically injecting code*, *graphical visualizations of data*, *create new graphical visualizations during runtime* and their *intent*, i.e., what do they really try to address.

2.2.1 Scala Worksheet

Scala Worksheet is a Scala plug-in, available for IntelliJ and Eclipse, for multi-line REPL. A simple worksheet implementation, it is a glorified editor with the option of evaluating the script and placing the results of each expression in a comment on the same line. Scala worksheets are normal Scala files, except that they end in *.sc* instead of *.scala*.

Users start by creating a Scala Worksheet project through project creation wizard. Once finished, they begin typing and notice that most features of the Scala editor are available. Whenever users save their code, the worksheet is re-evaluated and the results are printed in line with your program.

Figure 2.1 shows an example of such mechanism.

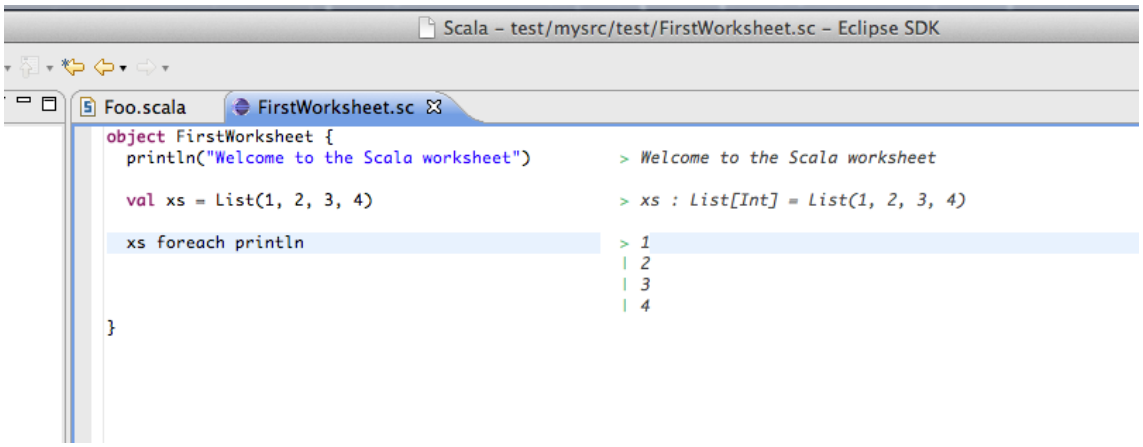

Figure 2.1: Screenshot of a worksheet being re-evaluated and printing results

Scala Worksheet offers a live development environment in which users see their code being evaluated automatically without need for compiling and executing it.

These worksheets served as a inspiration for the liveness traits of Visual Scala, however, a command-line approach was taken instead of a file editor. Also, it only offers a verbose output of instructions whereas in Visual Scala, type values have many graphical representations. Table 2.1 shows the analysis regarding this tool. Scala Worksheets provide dynamic injection of code when running, do not provide graphical output on data structures and although possible to create conversions they would not be rendered.

Scala Worksheet	
Dynamic injection of code	*Yes*
Graphical output on data structures	*No*
Create new conversions	*Possible because it is for Scala, but would not be rendered as graphics*
Intent	*Tool designed for simulating a live environment for Scala where code is evaluated while the user is typing*

Table 2.1: Feature analysis on Scala Worksheet

2.2.2 Simply Scala

Simply Scala is an online Scala interpreter available at `http://www.simplyscala.com/`. It acts simply as a *bridge* to a Scala interpreter. Users can code in it as they code in a typical Scala REPL.

The goal for Simply Scala was for people to jump in and start learning Scala, providing lots of examples and immediate feedback.

The whole application is written entirely in Scala, even the frontend HTML is generated using Scala's XML literals. It runs on Google App Engine, and the client UI is written in JavaScript.

Simply Scala resembles Visual Scala in the way that it offers a web application to provide a *bridge* to a backend Scala interpreter for user code evaluation. However, Simply Scala offers no additional features, it is simply a Scala interpreter on the web.

Figure 2.2 shows a screenshot of the Simply Scala web page and the results from interpreting some Scala instructions.

Table 2.2 presents the analysis on Simply Scala. Simply Scala offers dynamic injection of code during runtime, because it is a Scala interpreter, does not provide graphical representations of data structures and although possible to create conversions they would not be rendered.

2.2.3 Scripster

Scripster is an interactive scripting pad for Scala. It is an web application offering a GUI component for typing in Scala code and evaluate it. It doesn't follow the protocol of an interpreter, in which expressions are submitted, one at a time, for evaluation. The user writes Scala code as they would in a file and request lines of code to be evaluated using command shortcuts.

Scripster is available at `http://www.tryscala.com/`.

Figure 2.2: Screenshot of the Simply Scala Web page with results from interpreting Scala code

As far as the feature analysis on Scripster, it proved the "worst" tool found for Scala development. It does not provide dynamic injection of code, it offers a code editor area, where lines or selections of code can be triggered to be evaluated. Graphical visualizations on data structures is not possible as well as creating some. Table 2.3 shows the analysis result on Scripster.

2.2.4 The Julia Language

The Julia Language is a new dynamic language for technical computing, designed for performance from the beginning by adapting and extending modern programming languages techniques.

The goal of Julia is to take advantage of certain modern techniques for executing dynamic languages in a more efficient way. They claim that Julia has the performance of a statically typed language while still providing interactive dynamic behavior and productivity like Python, LISP or Ruby [BKSE12].

It is implemented with JIT compilation using the LLVM compiler framework [LA04], making available to the programmer a type system that still remains unobtrusive in the way that the programmer is never required to specify types.

Figure 2.3 shows an example of generating mathematical plots representations with the Julia Language.

When analyzing Julia, it proved one of the most influential tools for Visual Scala. Julia offers dynamic injection of code as it is an interpreter-like approach, with a continuously live state, offers graphical visualizations on data structures but does not provide ways for creating new ones. Table 2.4 shows the summary of Julia's analysis.

Simply Scala	
Dynamic injection of code	*Yes*
Graphical output on data structures	*No*
Create new conversions	*Possible because it is for Scala, but would not be rendered as graphics*
Intent	*Web application designed to be an online version of the default Scala interpreter*

Table 2.2: Feature analysis on Simply Scala

2.2.5 Online Python Tutor

Online Python Tutor is a web-based program visualization tool for Python, which is increasingly becoming a popular language for teaching introductory computer science courses [Guo13]. It enables programmers to step forwards and backwards through the code execution to view the run-time state of data structures and also share their program visualizations on the web.

When teaching the execution trace of an algorithm, usually instructors tend to draw on white-boards, which can be tedious and error-prone [OVZS12], the steps from the execution with the associated state of data structures. This helps students to better visualize how data structures change throughout execution and begin to fully understand computation.

Online Python Tutor's GUI consists of a code editing area, where users type in Python code, and a set of control buttons to step forwards and backwards on the execution trace. Once the code is correctly written, the execution trace is available for navigation. When navigating, to the right side of the code editing area, there is a GUI component which shows the current state of the data structures declared in the user code.

Figure 2.4 shows a snapshot of a certain step in the execution trace of a Python algorithm that calculates, in a recursive way, the sum of a linked list.

Although similar, this project differs to ours in the sense that it offers graphical visualizations over the execution of a certain algorithm or piece of code. The environment is not live, the entire code is submitted for evaluation and debugging at once, not offering a way to evaluate code similar to a REPL environment.

Online Python Tutor is free and open source software, available at `http://pythontutor.com/`.

This Python Tutor is quite a powerful web application, although not offering dynamic code injection, it offers great graphical visualizations on data structures and how they are changing throughout the code's execution. Users cannot create new representations of data structures as they are intended to be visualizations as one would draw on a whiteboard when trying to explain or teach computation. Table 2.5 shows the summary of analyzing Online Python Tutor.

Scripster	
Dynamic injection of code	*No*
Graphical output on data structures	*No*
Create new conversions	*Possible because it is for Scala, but would not be rendered as graphics*
Intent	*Web application designed to be an online code editor for Scala, evaluating lines or chunks of code*

Table 2.3: Feature analysis on Scripster

2.3 Summary

From researching the state of the art on this dissertation' context, we've learned that *Integrated Development Environments* are very mature tools and capable of providing many improvements on programmer's productivity. However, these tools apply best for statically typed languages where user code is compiled and then executed. With new additions of code having to be re-compiled and executed again. And generally these tools don't support dynamic execution of code during *runtime*.

As to *Live Development Environments*, the concept is still growing and eventually we will have highly capable and mature tools for programming with dynamic languages. With a good example in Adobe Brackets, for instance. We've also learned that some tools for development, implement dynamic execution of code by means of an interpreting approach, like a simple REPL does.

On another topic, some of these tools provide graphical representations of data structures, which is the main focus of this dissertation. Normally associated with more technical or mathematical computation, these tools enrich the visualization of data by powerful graphics and widgets. These visualizations are however, explicitly invoked by the end-user, which is not the case for Visual Scala, where the goal is implicit conversions.

We can safely conclude that no tool exists that does exactly what Visual Scala aims to do. There are environments which convert data to graphics, but not in an implicit way. They also don't allow for new conversions to be created, whereas Visual Scala does.

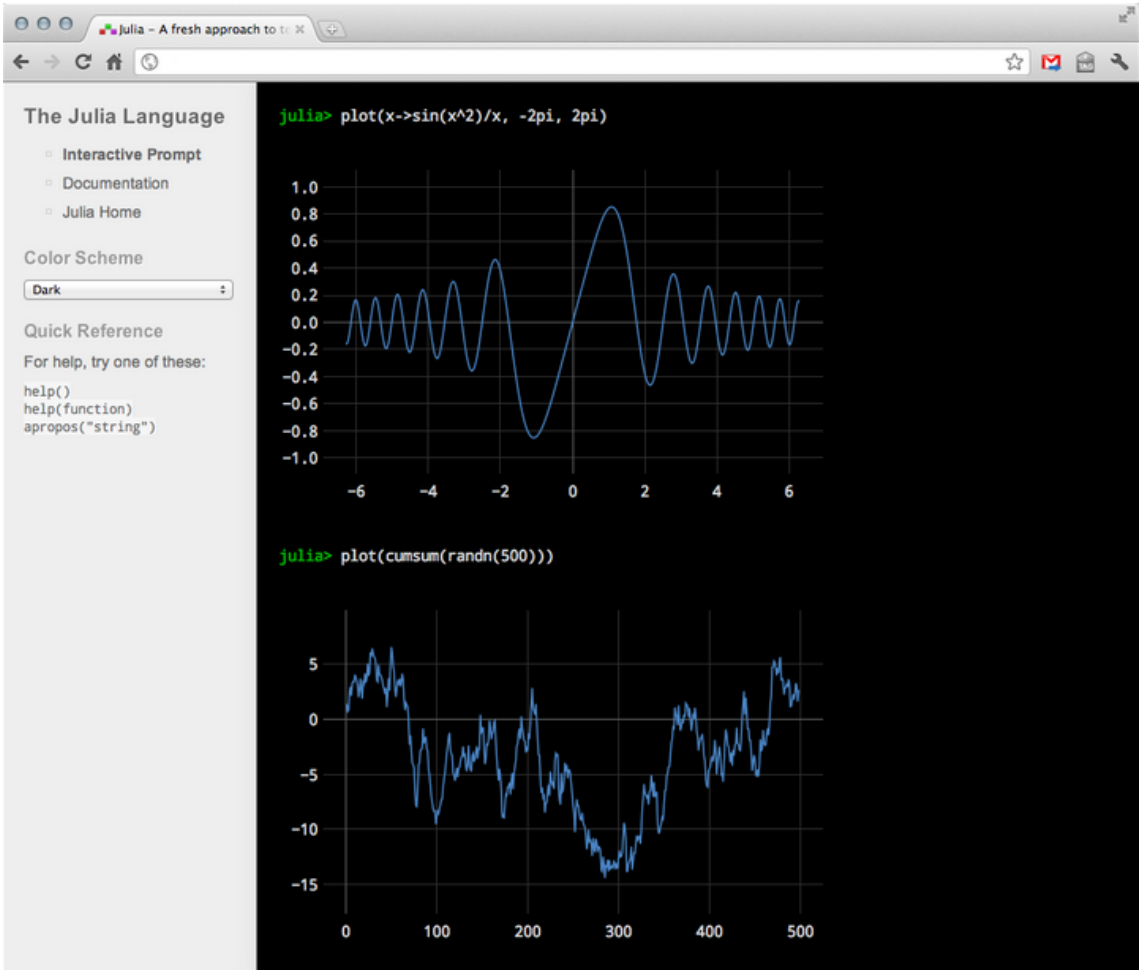

Figure 2.3: Example of using the Julia Language to visualize mathematical plots

The Julia Language	
Dynamic injection of code	*Yes*
Graphical output on data structures	*Yes*
Create new conversions	*No*
Intent	*Dynamic language for technical computing, offering an interpreter-like component for entering code and see it render graphically*

Table 2.4: Feature analysis on Julia Language

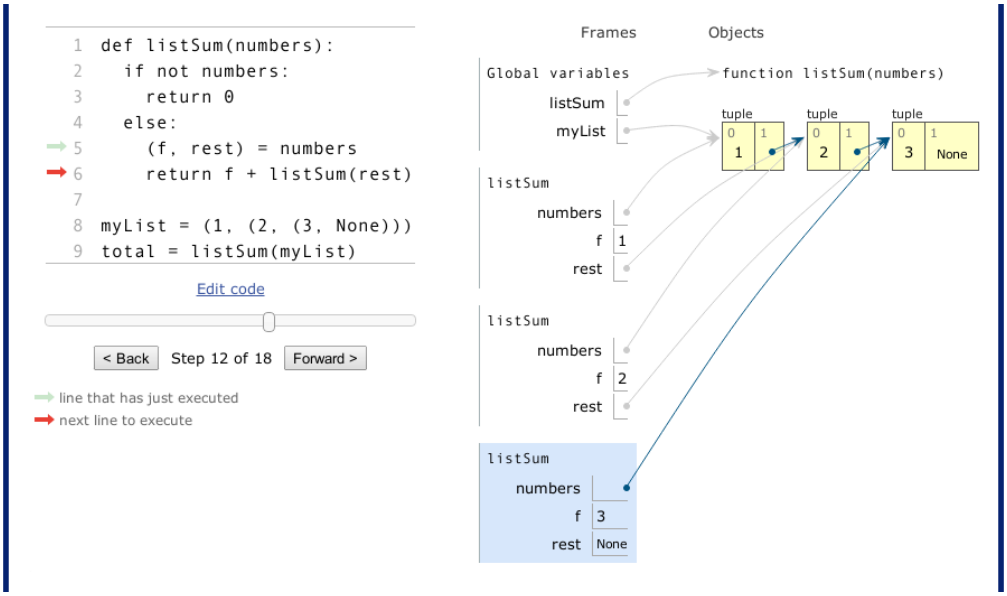

Figure 2.4: Screenshot of an execution trace from Online Python Tutor

Online Python Tutor	
Dynamic injection of code	*No*
Graphical output on data structures	*Yes*
Create new conversions	*No*
Intent	*Powerful web application for visualizing the execution trace of Python code. Data structures have graphical visualizations and they change as we iterate through the code's execution*

Table 2.5: Feature analysis on Online Python Tutor

Chapter 3

Designing Visual Scala

The main goal is to explore the flexibility of Scala's implicits towards an extensible live environment. In order to do so, successfully implementing and simulating such an extensible and live environment was necessary so we could, in fact, explore the flexibility of Scala's implicits.

The current chapter tours over the design process that took place before implementing Visual Scala. This chapter will be covering the necessary requirements as well as a higher-level architecture and several decisions when tackling some issues. Although it might seem that this process followed a linear approach to development starting with designing, requirements outlining, implementation, etc., a circular iterative process in which all development phases were repeated as many times necessary to correctly refine and optimize Visual Scala.

3.1 Requirements

At the starting point for almost every software development are requirements. Key traits and features Visual Scala should have were discussed and outlined and the resulting requirements were grouped into two categories: *functional* and *usability*. Functional requirements refer to those that are part of Visual Scala's *core*. They are absolutely essential because as the name suggests, they provide *functionality*. Usability requirements refer to those specific to the usage of Visual Scala, they define overall features and interaction of the end-user with the system.

3.1.1 Functional

At the core of Visual Scala are important requirements like:

- **Dynamic evaluation** of user code during runtime of the system;

  We want to the development environment to be live, and as to achieve it we needed to dynamically execute user code during runtime of the system. Some tools researched provide this functionality, like *Simply Scala*, *Scala Worksheet* and *The Julia Language*.

- **Extensible** resource libraries of graphical conversions;

We want Visual Scala's conversions library to grow easily with the help of the community. By placing the conversions library on the resources of the web server, new additions or changes do not represent a new re-compilation and initialization of the system. We want to provide an easy way for end users to send new conversions that they think are worthy of being part of the system. We found no evidence that state of the art artifacts allow for this contributions.

- **Convert** Scala's type values to HTML visual representations;

  We want Scala's native type values and data structures to have an implicit conversion to an HTML representation. Since there is no product that offers visual representations of Scala's types, this trait distinguishes our project from the rest.

- **Create new conversions** during runtime, without the need for restarting;

  We want Visual Scala to allow for new conversions to be created by users during runtime. Either by using a form which helps users to create them or by directly submitting code for new conversions, we want to extend the conversions library throughout Visual Scala's usage without the need for re-compilations or restarting the system. No related tool was found that provides this functionality, making our project innovative.

- **Online REPL** in order to target more users and avoid installations and download times;

  We want a development environment that can target many users and avoid them having to download and install Visual Scala. The web offers a portable environment ensuring that every user is using the most up-to-date system, allowing as well for easier sessions sharing and cooperative usage. Other researched tools offer an online environment, like *Simply Scala*, *Online Python Tutor*, *The Julia Language* and *Scripster*.

- **Multiple users** accessing the web server;

  As to provide an individual evaluation session for an end user, we need Visual Scala to allow access for multiple users. By instantiating a new interpreter session for each user, other user's code will not interfere with each other's environment state. Almost every tool found that is web based offer this functionality, being a standard requirement.

They seem like few requirements but these are the ones considered *core features*, from which everything else is constructed and derived.

3.1.2 Usability

From an usability standpoint, several requirements were also discussed and outlined in order to begin to fully envision the GUI component for interacting with Visual Scala's backend. And so, as a user I must be able to:

1: *Submit code for evaluation*; **2:** *Interact with the system in a command-line fashion*; **3:** *Whilst typing, hitting Enter should insert a newline if the instruction is not complete*; **4:** *Navigate*

through the history of submitted instructions with the up and down arrow keys; **5:** *Collapse and show the graphical visualization of a result*; **6:** *Keep an history of previous results on the output body*; **7:** *See the list of available conversions for a certain result type*; **8:** *See all active conversions*; **9:** *Edit any source file of the conversions library*; **10:** *Revert changes made on those source files*; **11:** *Create new conversions during runtime*; **12:** *Refresh the web page and keep previous results*; **13:** *Possibly interact with graphical visualizations and create new values from changes*.

3.2 The approach

In the previous section the reader learned about the requirements that were established for the implementation of Visual Scala. Besides requirements, once a more general view of what needed to be done was clearly set, settling several decisions needed to be done. These decisions range from what should be the high-level architecture of the system, communication protocols, technologies to be used, programming languages, etc.

3.2.1 Methodology

In section 1.2 we introduced some research methodologies. Regarding this dissertation's nature and hypothesis we followed an *Engineering Method*, where we developed a solution to test a set of hypothesis. The implementation process followed an interactive and incremental methodology, in which the solution was refined and re-designed/re-developed in order to better meet the dissertation' goals.

3.2.2 Decisions

Let's review some of the decisions made regarding the design of Visual Scala.

3.2.2.1 High-Level Architecture

As a way to target more users, avoiding downloads, installations or updates, Visual Scala was intended to be a web application. As a result, the environment must follow a client-server system model.

Client Server Model This model is an approach to computer network programming, being prevalent in computer networks. Widely used, Email, the World Wide Web as well as network printing all apply the client-server architecture principles.

The goal of this model is to assign two roles to computer/nodes of a system: client or server. A *server* acts as a system that aims to selectively share resources, perform computation or other services. A *client* acts as a system which goal is to initiate contact with the *server* and request resources, computation, etc.

The Client-Server model allows for multiple clients to access the same server at the same time.

The two-way exchange of messages/data is a *request-response messaging pattern*. Clients and servers exchange *request* and *response* messages via, usually, HTTP requests through a network.

In light of this model, every browser acts a client and our Scala backend as the server.

Figure 3.1 depicts the system's parts and how they interact.

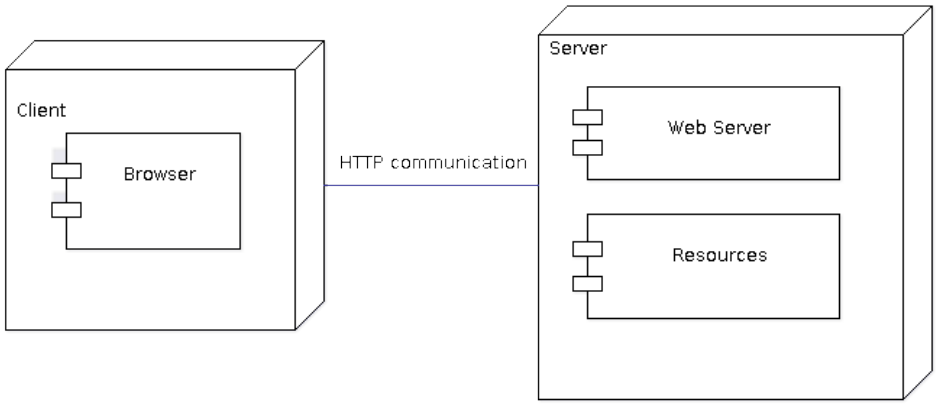

Figure 3.1: Visual Scala's high-level architecture

In the following sections we'll present some decisions regarding both nodes of Visual Scala's system.

3.2.2.2 Backend

As the goal is to explore flexibility of Scala's Implicits towards an extensible live environment, the programming language to implement the system's backend was quite clear, Scala. There was no need for implementing the Web Server in another language and then to invoke Scala modules for evaluating user code and the Conversions mechanism. And so, everything in the backend is implemented with Scala.

As a key component of every client server model, it is necessary to implement a Scala web server to receive and send HTTP requests. Building one from scratch was not an option and so a decision on which Scala web framework to work with was necessary. They were several options available, like Scalatra, the Play Framework, Unfiltered, Bowler, Socko Web Server, among many others. Lacking previous experience with Scala, some experiences were made with different web frameworks but Unfiltered (You can read more about Unfiltered in section 4.6) was the choice due to its obvious simplicity. So, the web server was to be implemented using Unfiltered.

What about dynamic user code evaluation? Would we write a compiler/interpreter for Scala? There was no actual need for this, since there are already available implementations for evaluating Scala code. At first Twitter's Eval library was considered a viable option, which offers an extensive

API for dynamically evaluating Scala code. We tried it, but results lacked behind expectation and it lead us to trying out a smaller GitHub project named ScalaInterpreterPane (You can read more about ScalaInterpreterPane in section 4.6), that offered an API for forking standard Scala's interpreters with an abstraction layer for manipulating results from interpretations. Despite its limitations, as it does not provide a way of evaluating several expressions and obtaining result objects for each result produced, it was quite simple to use and provided a nice way of evaluating user code. And so, forking new instances of the Scala interpreter and dynamically evaluating user code is done using ScalaInterpreterPane.

Upon tackling the most important features of Visual Scala, the design of Visual Scala's back-end architecture began its process. Visual Scala's presents a quite simple and small architecture. It is composed of three main components, namely, the Web Server, the Evaluation module and a Miscellaneous module for general purpose procedures. The conversions library is part of the system's resources as it is not part of the *running* backend. Each component's implementation will be discussed in chapter 4.

Figure 3.2 shows the architecture of Visual Scala's backend with the typical flow of interactions between the different classes when evaluating user code, the typical interaction.

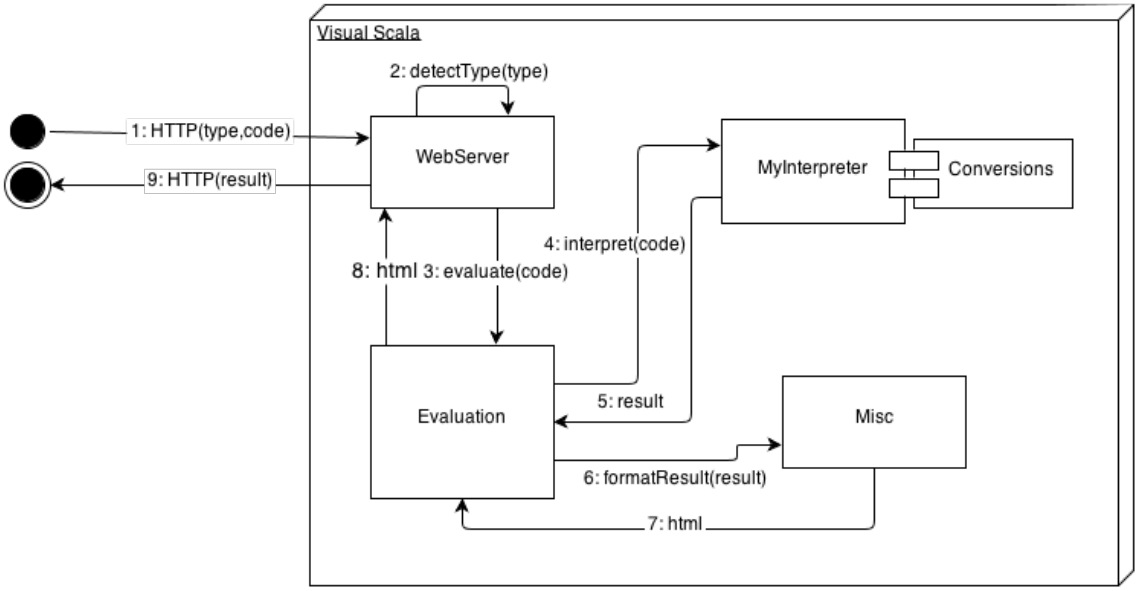

Figure 3.2: Diagram representing Visual Scala's backend architecture with the typical flow of interactions

3.2.2.3 Frontend

The backend for our system was to be implemented in Scala, but what about the frontend?

Earlier from the start we wanted Visual Scala to be a web application since there are many popular JavaScript libraries that enable more convenient data manipulation [jQua, Pro13, too13, Wic09]. By supporting HTML, CSS and JavaScript libraries we believe we can power the creation of graphical representations of data more easily than graphical Java libraries like Swing. It also

makes it easy for users to try it out since no installations or regular updates are required. With a simple way to integrate new conversions onto the environment and with a support for different existing graphical libraries we believe the community can easily expand the system.

Web applications can create a bigger impact on the community and Visual Scala will walk amongst the World Wide Web.

Regarding the design of the GUI component, at first, inspired by Scala Worksheet (You can read more about Scala Worksheet in section 2.2), we wanted to have a file editing interaction, where end users would write in a code editor area and the evaluation would take place automatically. However, regarding issues of detecting starting and ending points of expressions and other issues associated with continuous evaluation, we felt it was best to approach a command-line fashion. By simulating a typical *Interpreter* environment, in which users type expressions one at a time, we achieved faster results towards implicitly converting data structures and Twitter's Bootstrap (You can read more about Twitter's Bootstrap in section 4.6) made it easy and elegant to implement the GUI component of Visual Scala.

3.3 Summary

In this chapter the reader had a chance to read about the design process of Visual Scala. From requirements to high-level architecture and to specific decisions regarding both frontend and backend were also discussed.

The system is rather simple as it does not present a complex architecture. The idea was pretty much straightforward, to develop a *bridge* to a background Web server running standard Scala interpreters to evaluate user code.

Regarding the Scala server, apart from evaluating user code as is, only a component for acting as a type value's conversions library was to be designed. The main issues relied on allowing for such a conversions library to be part of the running interpreter, and thus, be extensible while the user is playing around with it.

Liveness was easily achieved following an command-line approach which made easy the use of the default Scala interpreter to evaluate user code and produce output.

Chapter 4

Implementation

The previous chapter described the requirements and the design process of the project. The current chapter will detail the implementation process of the dissertation' project. Elements such as the system's architecture, API, protocols, etc, will be presented and described.

The project followed an incremental process with several approaches that were tested regarding the Web app frontend. However, only the final solution will be target of discussion.

We'll start by presenting the high-level architecture of Visual Scala and detailed descriptions of both frontend and backend components of the system. Section 4.2 will explain how the frontend was implemented as well as an overview on its GUI component. The backend node, its architecture and detailed description of each of its component will be presented in section 4.3. The conversions library explanation, how Scala's implicits work and how the conversion are *actually* implemented are covered in section 4.3.3 and 4.4, respectively. We'll continue on by listing the implemented conversions on some Scala's type values in section 4.5 and finish this chapter by presenting the technologies used, in section 4.6.

4.1 High-Level Architecture

The system's architecture follows a simple client-server approach, as previously discussed in section 3.2.2.1, having different clients accessing resources from a Web server.

Every browser acts as a single client, issuing requests to the Web server, be it the main page or code evaluations and the respective obtained result. From here on out, we will be referring to the Server as the *Backend* and to the Client as the *Frontend*.

4.2 Frontend (HTML/CSS/JavaScript)

The frontend, or a client, of this system consists of any device with an active network connection and a browser. The browser issues a *GET* request of the main page from the Web server and it engages on an active Scala development environment session.

Figure 4.1 represents the main page obtained when the browser connects to the Web server and an instance of a new Scala interpreter is properly configured and running.

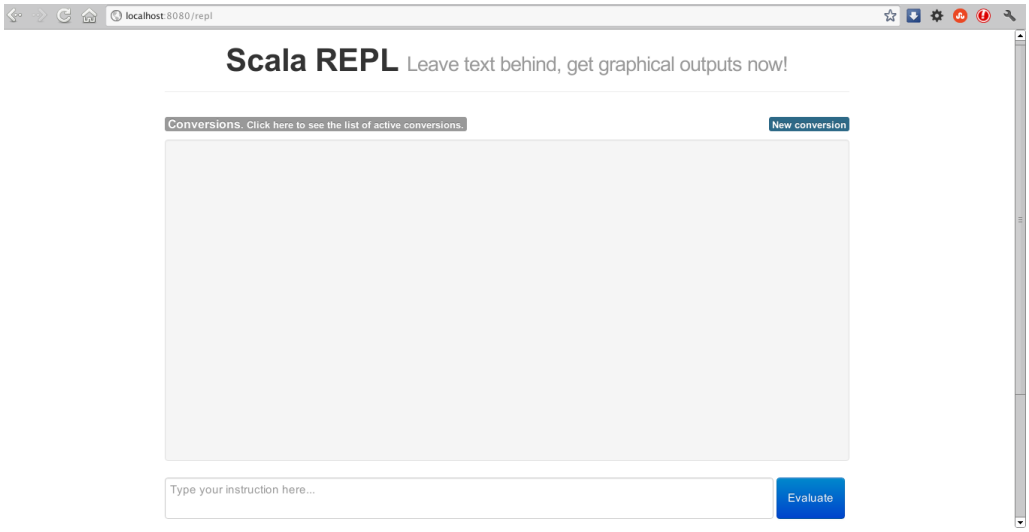

Figure 4.1: Screenshot of the main page of the Extensible Live Development Environment

A typical workflow of the web application consists in connecting to the web server and requesting an active session on a Scala development environment. The user then types Scala code instructions, as they would on the standard Scala's interpreter, and submit the code for evaluation. If the code is correct, a result shall be appended to the main output body with information regarding the instruction provided and the associated result, as well as a list of possible conversions on that same result. Code evaluation request are issued through AJAX *POST* HTTP requests.

Figure 4.2 shows the main page with obtained results from evaluating user code.

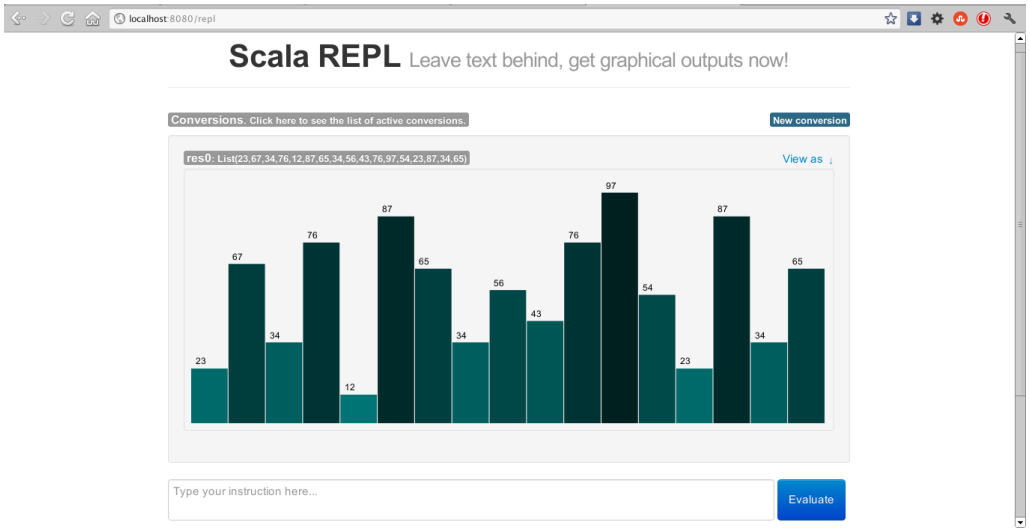

Figure 4.2: Screenshot of results obtained from interpreting user code

In the example depicted by Figure 4.2 we see that the user created a `List` with numerals in it and is currently viewing it's value as a graphical bar chart. Apart from the current result's graphical

visualization the user can also check the list of other conversions on the result's associated type value by selecting the top right option *View as*.

Apart from normal code evaluation procedures, similar to a default Scala interpreter usage, the user can also see a list of the active conversion source files which were loaded onto the interpreter. The user can then see the contents of those files, edit them if they desired and submit the changes. In case some previously made changes are not desired, the user can revert a source file to it's original content by clicking on the respective button.

Figure 4.3 shows the list of conversion source files and manipulation of related contents.

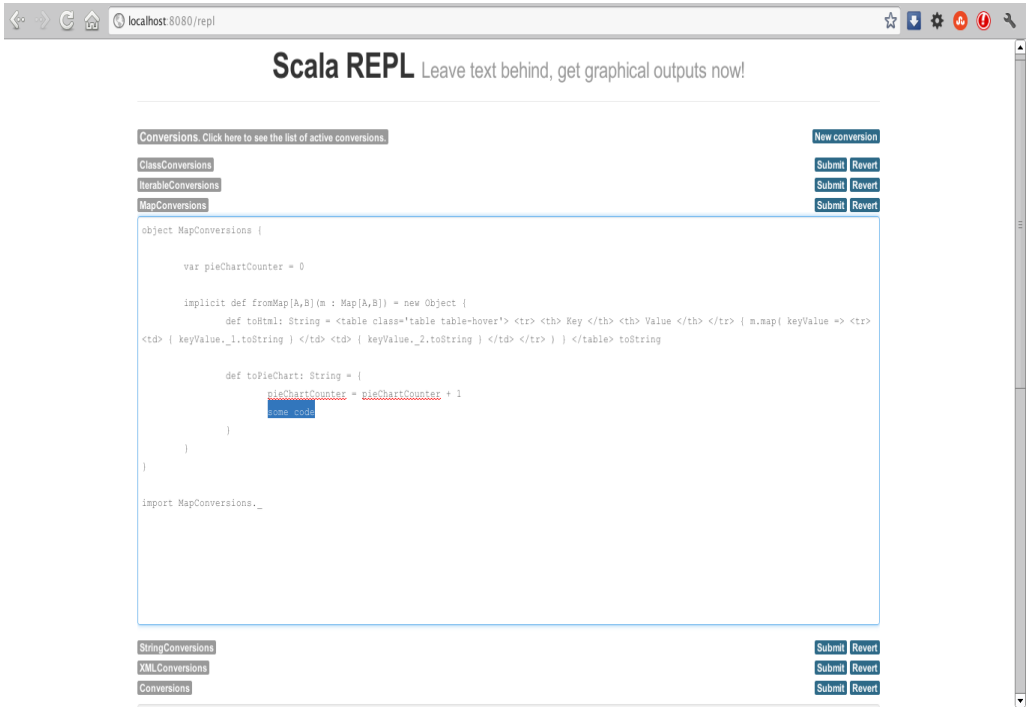

Figure 4.3: Screenshot of inspecting source files of the conversions library and edit its contents

Since the conversions library are Scala source files that are loaded onto a interpreter, new conversions can be created through the use of a form, available from the option of a new conversion. The user is presented with a modal form with the necessary fields to be filled with the conversion code and type value that applies to. If correct, the conversion is loaded onto the interpreter and a new option become available on the conversions list for converting the specified type value to the new conversion.

Figure 4.4 shows the modal form for creating a new conversion.

4.2.1 Cookies

Having a single Scala interpreter instance, running on Visual Scala's backend, to evaluate all user code was not feasible. We want multiple users to access our Scala backend and each one to have their own evaluation session, as to say, their own interpreter instance.

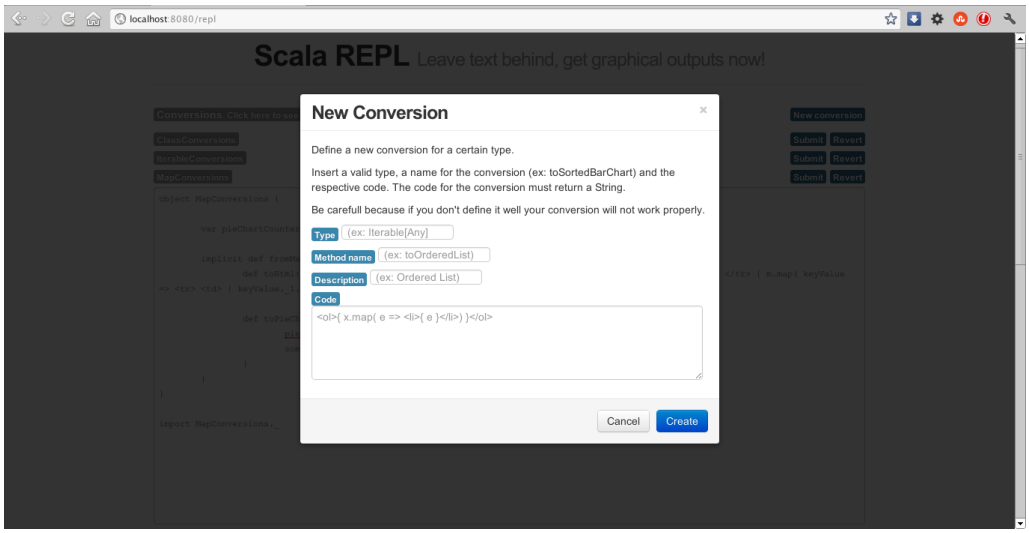

Figure 4.4: Screenshot of the form available for creating new conversions

To address this, we simply need to create multiple instances of interpreters, but how do we know which instance corresponds to their *owner*? As a standard mechanism used throughout most of the Web, we can identify users by using Cookies.

Cookies, or Web Cookies, are small pieces of data sent from a website and stored in a user's web browser. They were designed to be a reliable mechanism for websites to remember the state of the website or activity the user had taken in the past. Cookies contain relevant information to be used by the backend web server and usually travel back and forth between the web server and the user's browser.

And so, we identify each user by a *session* variable containing a `String` representation of a `Double` numeral and send that information with Cookies. Session keys are generated randomly by the Scala web server and stored on the user's cookie information. So, every AJAX request, with user code that is sent for evaluation, comes with a Cookie containing the user's session key which corresponds to a specific Scala interpreter instance.

4.3 Backend (Scala)

The Server side of this architecture consists on a Scala Web server. This web server is composed of essentially three parts which are: the Scala web server managing requests and responses, an Evaluation module responsible for evaluating user code and the Conversions library responsible for storing all implicit conversions to be used.

The backend's architecture can be visualized in Figure 3.2 in the previous chapter 3 and we'll continue on by describing each component and their relations.

4.3.1 The Scala Web Server

Implemented using Unfiltered[1], a Scala Web Server is endlessly listening for HTTP requests on its associated domain. This component is responsible for managing HTTP requests as well as cookie information. The server manages all operations from inspecting the user's cookie information about already pre-established interpreter sessions, to establishing new Interpreter sessions as well as relegating user typed code for evaluation.

The server is listening for *GET* requests on different domain paths such as, */* or the root domain, */scala* and */repl*. Essentially, all domains, besides */repl* are forwarded to the */repl* path, and from there, a new interpreter is instantiated, or not (if it was already done previously). The server responds with the main page depicted in Figure 4.1.

The server is also listening for *POST* requests on the */repl* domain with user typed code. The code is received and the web server delegates the code to the Evaluation module in order to be evaluated by the correct interpreter instance (corresponding to the user's session key). Once the evaluation is complete, the server encapsulates the value returned from the interpreter in an HTML structure to be appended to the main output body of the main page. An example is shown in Figure 4.2 regarding a result from interpreting a user's instruction.

4.3.1.1 Communication & Messages

There are several possible interactions with the Scala web server resulting in different procedures to be triggered. Some are directly related to evaluating user code and others are just requests of contents of the conversions library. Clients and server need to speak the same *language*, and thus a message structure was defined as to indicate all possible interactions with the server.

Whenever the client issues an *HTTP* request to the server, a *JSON* object is constructed with the following structure:

```
1 {
2    type: TYPE_OF_INTERACTION,
3    code: CODE_REFERRENT_TO_TYPE
4 }
```

Both fields are encoded as `Strings` and the **type** can include the following options:

1. **EVAL** "Eval" refers to the standard case where user code is to be evaluated and wrapped around an HTML element for integrating the result list on the frontend. **Code** contains the user code. The next example illustrates a possible EVAL request.

```
1 {
2   type: "Eval",
3   code: "val x = 2"
```

[1] You can read more about Unfiltered on section 4.6.1.1

```
4 }
```

2. **PARTIAL** "Partial" indicates that a conversion is to be applied to an already existing identifier (or result). This conversion is not wrapped around an HTML element and it is sent over for replacing a conversion on a previous result. **Code** contains the invocation of the conversion on the desired identifier, like `res0.toBinaryTree`. The next example illustrates a possible PARTIAL request.

```
1 {
2   type: "Partial",
3   code: "res0 :!: toBinaryTree"
4 }
```

3. **REQ** "Req" refers to requesting the contents of some file from the conversions library. Whenever the client requests the main page, it also requests every file's content of the conversions library. This is done because the user can edit the contents of the files on the frontend and submit the changes for evaluation. **Code** contains the name of the file to be read. The next example illustrates a possible REQ request.

```
1 {
2   type: "Req",
3   code: "IntConversions"
4 }
```

4. **UPDATE** "Update" refers to the procedure of submitting changes to a conversions library file. From the contents loaded with REQ, the user can edit them and submit the file for re-evaluation when desired. **Code** contains the modified contents of the conversions file to be re-evaluated. The next example illustrates a possible UPDATE request.

```
1 {
2   type: "Update",
3   code: "object IntConversions {
4     implicit def fromInt(i: Int) = new Object {
5       def toHtml = <h2>{ i }</h2> toString
6     }
7    }
8    import IntConversions._"
9 }
```

5. **REVERT** "Revert" refers to reverting the changes previously made on some conversions library file. If the user altered the file via UPDATE and no longer wants those changes, it

can revert to the original file. The file is re-read from the library and re-evaluated. **Code** contains the name of the file to be reverted. The next example illustrates a possible REVERT request.

```
{
  type: "Revert",
  code: "IntConversions"
}
```

6. **NEW** "New" refers to creating a new conversion. In this case, because there is no need to construct HTML elements or anything, the conversion is evaluated and if successful the server responds with simple *SUCCESS* or *FAILURE*. **Code** contains the conversions code to be evaluated. The next example illustrates a possible NEW request.

```
{
  type: "New",
  code: "implicit def conversion1(x: Double) = new Object {
      def toHtml = <h1>{ x }</h1> toString
      }"
}
```

4.3.2 Evaluation Module

Essential to achieve our main goals, Scala's Read-Eval-Print-Loops, or Interpreters, are necessary to evaluate user code. Whenever the Server receives a new session key, or a new connection, the Evaluation module instantiates a new Interpreter object to be used for evaluating the code for that specific session.

The interpreter object is injected with the graphical conversion library so it can be ready for converting type values to graphical visualizations.

Interpreters are created using ScalaInterpreterPane, a GitHub project offering an API to create Scala's Interpreters with an abstract layer on top for managing evaluation results. You can read more about ScalaInterpreterPane further on section 4.6.1.2.

4.3.2.1 Standard Evaluation Process

Inside the Evaluation module, every Scala expression undergoes an evaluation process comprised of different steps. Upon receiving a *POST* request with user typed code, the Web server first inspects the type of the request to know what interaction is necessary. The possible types are discussed in section 4.3.1.1, please refer to it if any doubts arise. Regarding to evaluation, three cases may occur, one for a user code evaluation with HTML response, one for converting an existing result and another for evaluating new conversions with no responses.

For the first and most common interaction, which is type **Eval**, the evaluation process is not straightforward. Once user typed code is received on the Web server, is it sent over to the Evaluation module which will interpret the code and return a result.

Three different phases define the evaluation process and are represented in Figure 4.5.

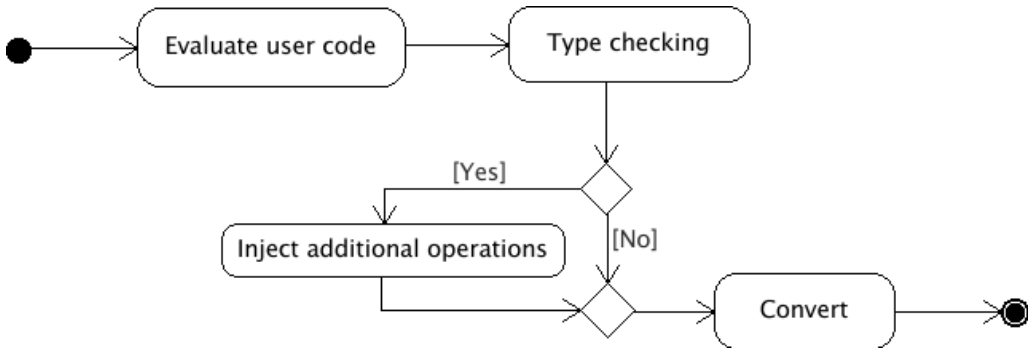

Figure 4.5: Diagram representing the different stages of the evaluation process

Phase One starts out by simply interpreting the user's code as is. The code is submitted to the target interpreter and evaluated by using ScalaInterpreterPane's API. We then gather the result's identifier and move on to phase two.

Phase Two in which we inspect the result type and if it matches some pre-defined types we need to apply additional operations to it. As an example, if the obtained result is an instance of a Scala `Iterable`, we want to slice it. Since the default `toHtml` conversion for Iterable's is a paginated list, we don't want to show a big number of elements and so we slice the Iterable to a fixed, pre-established size.

After that or if no pre-conversion operations are needed for the type value of the result we move on to phase 3.

Phase Three corresponds to the last phase, in which we invoke the `toHtml` method on the result's identifier. The invocation on the identifier from Phase 1 or 2 is submitted to the target interpreter via the ScalaInterpreterPane's API. The conversion result is sent over to the Web server which will, in turn, encapsulate it in an HTML element and send it over to the user's browser.

4.3.2.2 Conversion Process

The HTML element that encapsulates the result from the standard evaluation contains a list of other possible conversion for that result. Say, `res0` has a conversion option called *to Binary Tree*. When the user explicitly asks for that conversion (using the dropdown list *View as*) the frontend issues a request of type **Partial**.

The Web server detects the type **Partial** and that this results in a conversion for an existing result and invokes the code received. Since it is a new visualization for a previous result, there is no

need to build a full response to be appended to the main output body on the client's page. Instead, only the graphical visualization code resulting from invoking the desired operation is returned in order to replace the previous visualization on that same result.

4.3.2.3 New Conversion

New conversions are represented by the type **New** in the messages received by the Web server. Although designed for new conversion, this behavior is used for either new conversions or new changes on conversions files. This is because the main aspect of such a procedure is to submit code for evaluation on the target interpreter and produce no HTML responses. The goal is to simply submit code for evaluation and if successful, reply with *SUCCESS* or *FAILURE*. The frontend only needs to know if such an operation was successful or not so it can inform the end-user.

4.3.3 Conversions Library

The Conversions library contains Scala source files that define different graphical conversions for Scala type values. This library is the most important component of the project, because without it it would just be an online web version of a typical Scala interpreter, just like Simply Scala. Each source file should contain only the conversions for a specific type value. This fragmentation is important to separate conversions so the end user can select which source files to load onto the interpreter.

This library is easily extensible by editing the existing source files or by adding new ones. The library is part of the backend system's resources and thus, there is no need to re-compile new changes in the sources.

A conversion file should follow a specific structure, explained further down. Figure 4.6 shows a draft of a valid conversion file.

```
object MapConversions {

    implicit def fromMap[A,B](m : Map[A,B]) = new Object {

    def toHtml: String = {
        <table> <tr> <th> Key </th> <th> Value </th> </tr>
        { m.map(keyValue => <tr><td>{ keyValue._1.toString }</td><td>{ keyValue._2.toString }</td></tr>
        </table> toString
    }
    }
}

import MapConversions._
```

Figure 4.6: Example of a valid Conversion library source file for Map conversions

All conversion source files must declare a new object, with no special requirements for its name. The object must contain an implicit definition of a method, again with no restrictions to its name, that receives the value type for conversion, as a parameter. This method must return an object which will contain the definition of the different conversions to be applied. Although it has been presented now as an anonymous object with no specific traits or classes, in section 4.4.3 we will cover the necessary changes to add recursiveness in conversions. Regarding this

solution example, no recursiveness is implied in the conversion process, so a composite type like `List[List[Int]]` would not convert the inner lists.

The standard conversion applied by default is called `toHtml` and in Figure 4.6 we can see `toHtml` converting the Map to an HTML defined table. Any number of conversions can be included in the implicit method that will, later on, be presented to the user as possible conversions.

You will learn more about Scala's implicits and how conversions work on section 4.4.

4.4 Scala's Implicits

As part of the conversions library, one aspect worth mentioning and detailing is the mechanism which allows for implicitly converting a Scala type value into an HTML representation. In this section we'll start by explaining what Scala's *implicit* keywords are and what they do as well as how we used it to elegantly solve our problem at hand.

4.4.1 Scala's implicit conversions

From *Programming in Scala: A Comprehensive Step-by-step Guide*, written by one of Scala's creator, Martin Odersky [OSV08]:

> *There's a fundamental difference between your own code and libraries of other people: you can change or extend your own code as you wish, but if you want to use someone else's libraries, you usually have to take them as they are.*
>
> *Scala's answer is implicit conversions and parameters. These can make existing libraries much more pleasant to deal with by letting you leave out tedious, obvious details that obscure the interesting parts of your code. Used tastefully, this results in code that is focused on the interesting, non-trivial parts of your program.*

So, how do implicits[2] work?

Implicit conversions are often helpful for working with bodies of software that were developed independently, without each other in mind. Libraries have different ways of encoding a concept that is often essentially the same thing. Implicit conversions come to help reducing *explicit* conversions that are needed from one type to another. We'll take a look at a possible use case for implicit conversions.

Let's say you are implementing your own version of a `print` method called `printty` that receives a `String` and it outputs it somewhere. Also, let's say you want to call `printty` without having to explicitly converting certain type values to `Strings`. By using Scala `implicit` keyword you can define a method that converts one type value to a `String`. Like this:

```
1  implicit def intToString(x: Int) = x.toString
```

[2]There are implicit conversions and implicit parameters, this section will only cover implicit conversions as they were the only ones used.

In this example you are simply calling the `toString` operation on your `Int` parameter but you can do whatever you like inside it to achieve the type value you want as a result.

The previously defined method `intToString` implicitly converts any `Int` to `String` whenever you provide `Ints` to methods which require `Strings`. And so, if you went on defining an implicit conversion for every type value you want your `printty` method, you would simply need to call `printty(3)`, `printty(4.5)`, etc, and have your parameter be implicitly converted to a `String` without you having to call `toString` on every parameter.

This powerful Scala feature works as follows. Every time an expression, for example `x + y` doest not type check, then the Scala compiler might try and change it to `convert(x) + y`, `x + convert(y)` or a combination of both, where `convert` is an available implicit conversion. And so, if `convert` changes `x` into something that has a `+` method, then this can fix the type check inconsistency detected earlier.

Various rules apply to the correct use of `implicits`[3] [OSV08].

So, whenever the compiler sees an X, but in turns needs an Y, it will look for an implicit conversion that turn X into a Y. As in the previous example described earlier, you don't need to explicitly call for conversions as the compiler implicitly applies them for you, if they exist.

4.4.2 Visual Scala implicit conversions

In Visual Scala, the goal was to create implicit conversions for Scala's type values to HTML representations to be displayed in a web application. Providing just a single conversion for a certain type seems limiting and poor, so we wanted to have many possible conversions for the same type value to different HTML representations.

To achieve this, we followed a different approach as to simple implicit functions that convert one type value to an HTML representation. And as to fragment the conversions library into separate source files containing all conversions for a specific type value, we define Scala `objects` to contain our implicit conversions.

At every result from evaluating user code, before sending it over to the web application frontend, we invoke a `toHtml` method to convert it to HTML. This is the default implicit conversion on certain type values. Although the Evaluation module of our Web server calls this conversion explicitly, from a end-user perspective, this conversion is made implicitly. We could have done this in an entirely implicit way but for having multiple conversions for one type value, implicitness would end past the default conversion `toHtml`.

Let's take a look at an example of a set of conversions for Scala's `Lists`.

```
1 object ListConversions {
2    implicit def fromList[A](list: List[A]) = new Object {
3        def toHtml = <ul>{ list.map( x => <li>{ x }</li> ) }</ul>
4        def toOrderedList = <ol>{ list.map(x => <li>{ x }</li>) }</ol>
```

[3] You can learn all about implicits and their rules in the book *Programming in Scala: A Comprehensive Step-by-step Guide*

```
5     }
6 }
```

In this example, we start by declaring an object called `ListConversions`[4] which shall contain only one `implicit` function declared inside it. We make use of Scala's keyword `implicit` to define the conversion function `fromList[A]`[5]. This function is parameterized to accept different types of `Lists`, like `List[Int]`, `List[Double]`, and so on. If we wanted to restrict the conversion for certain sub-type of `Lists` we could do so by explicitly defining the values contained in such `Lists`.

Last, but not least, all there was left was to define one or multiple conversions for `List`. The implicit function returns a new object which will define our conversions. In the example, we have defined two conversions: the default `toHtml`, to be called by the Evaluation module, and a `toOrderedList` which converts the input `List` to an HTML ordered list.

Once the conversions objects for multiple type values are coded, the Evaluation module *injects*, as to say submits their code for evaluation, all conversion objects onto the interpreter session configured for the end-user. After that, it imports them to the interpreter's environment, like `import ListConversions._`. Once that is done, the Evaluation module can invoke `toHtml` or `toOrderedList` on `Lists` and obtain the desired conversion.

The power of Scala's implicits provided us with a an, we could say, elegant way of defining multiple conversions and for the Evaluation module to ignore the type value at hand and simply call `toHtml` letting the Scala's compiler find the appropriate `toHtml` to be applied.

4.4.3 Recursive Conversions

Sometimes is interesting and useful to provide recursive conversions on certain data types. Consider for example a `Map[Int,List[Int]]` which maps `Ints` to `List[Int]`, in this case, we might want to convert the map to an HTML table but instead of simply verbosely displaying the `Lists` in the table we may want to convert them as well. Also, we wanted to apply this recursiveness in an flexible manner, without explicitly searching for certain data types and re-define their conversions inside this one. This is not feasible and thus it is necessary to implement a mechanism for recursive conversions by only invoking `toHtml` on each value that is being inserted into the conversion. Consider the following example of such a mechanism:

```
1 implicit def fromMap[A,B](m: Map[A,B]) = new Object {
2     def toHtml = (table code...) { m.map(e => <tr> <td> { toHtml(e._1) } </td> <td>
          { toHtml(e._2) } </td> </tr>) } (more table code...)
3 }
```

[4] There are no restrictions for naming these objects. Any name can be used.

[5] Again, no restrictions apply for naming as long as the method takes the desired type value as a parameter.

If we were to implement the recursiveness like the previous example the Scala compiler would throw a compilation error saying that types `A` and `B` do not have the `toHtml` method, which makes perfect sense. And so, there is a need to, instead of returning an anonymous object like `(...) = new Object (...)`, to return an object that the compiler knows it implements the `toHtml` method. To achieve it, we can define a Scala trait for the returned object, like this:

```
1 trait HTML[T] {
2    def toHtml(obj: T): String
3 }
```

With this trait, we are creating something like Java's *interfaces*, in which we are stating that an `HTML` object must implement his version of the `toHtml` method. And so, we go on by defining a general invocation of the `toHtml` with the *implicitly* keyword. And of course, the new implicit conversion method would return this new object type, like this:

```
1 def toHtml[A: HTML](a: A) = implicitly[HTML[A]].toHtml(a)
2
3 implicit def fromMap[A: HTML, B: HTML] = new HTML[Map[A,B] {
4      def toHtml(m: Map[A,B]) = (table code...) { m.map(e => <tr> <td> { toHtml(e.
           _1) } </td> <td> { toHtml(e._2) } </td> </tr>) } (more table code...)
5 }
```

With this approach, the compiler does not throw any error and can safely call the `toHtml` method on inner values of the `Map`. As a result, if the `Map` was to contain, inside it, other `Maps` would be converted as well.

4.4.3.1 Recursive Conversions Limitations

When we want flexible and elegant recursiveness some limitations were found. On section 4.4.3 we exposed how implicit recursiveness can be implemented. We are talking about recursiveness that is not explicitly explored, one can simulate recursiveness by exhaustively perform pattern matching on every element of a composite data type and apply different conversions regarding its type. This works but it's not flexible nor elegant.

In section 4.4.3 we presented a solution that involves `traits`. This solution works quite well when we are only considering that data types will have only one conversion, the default `toHtml`. Every result of the implicit conversions yields an object that shares the `trait` we have stated earlier, for example `trait HTML`. In this trait we are declaring that an object of this type must implement a method called `toHtml`. With this we are able to recursively invoke `toHtml` on inner types of composite types. Consider the following example of such a mechanism.

```
1 trait HTML[T] {
2    def toHtml(obj: T): String
```

```
3  }
4
5  def toHtml[A: HTML](a: A) = implicitly[HTML[A]].toHtml(a)
6
7  implicit def fromList[A: HTML] = new HTML[List[A]] {
8     def toHtml(lst: List[A]) = <ul> { lst.map(e => <li> { implicitly[HTML[A]].
          toHtml(e) } </li>) } </ul> toString
9  }
```

With this example we are defining a `trait HTML` that defines object which implement the method `toHtml` which receives any type and produces a `String`. We move on to define how to call `toHtml` with the *implicitly* declaration and finally define the implicit conversion on `List`. This conversion yields an object `HTML` which implements `toHtml` as a unordered HTML list. As we are invoking `toHtml` on every inner value of our parameter `List`, if such a `List` were to contain other `Lists`, they would in turn be converted as well.

This is elegant and provides a short way of working around recursiveness. This approach has however, some disadvantages. It is necessary to define conversions for every type that could be contained in `List`, like `Int`, `Double`, `String`, etc., even if it is simply calling `toString`. One way around is to define the conversion for `Any` which maps to every Scala type value. This disadvantage results in a bigger amount of conversions code to be injected in the target interpreter, resulting in a higher loading time.

Another disadvantage arises when we want to define multiple conversions for the same type and not just `toHtml`. If we were to define just one trait as we did before, and add it another conversion, like:

```
1  trait HTML[T] {
2     def toHtml(obj: T): String
3     def toOrderedList(obj: T): String
4  }
```

This would result in having to define the `toOrderedList` on possible inner types, as for types like `Int` or `Double` this conversion does not make sense. Of course we could always overcome this by defining those conversions as simply calling `toString` on those types but it feels like redundant and unnecessary code. A different approach to this could be defining different traits for certain types, like `trait ListHTML` and `trait SimpleHTML` but we wouldn't be able to parameterize the parameters on the implicit functions.

4.4.4 Flexibility of Scala's Implicits

Visual Scala's conversions basically consists on functions that, when provided with certain inputs, generates `Strings` containing HTML code that, complemented with JavaScript, represent in

graphical manners, the provided input. Almost every programming language can implement such functions. But how to correctly apply each conversion function to specific native type values?

4.4.4.1 Conversions without Scala's Implicits

In possibly every programming language, there are conditional statements, usually referred to as *If then else* which provide ways of conditionally executing behavior when certain conditions are *true*, or *false*. Regarding such conditional control mechanisms, a conversion library can be built using *Ifs*. We would have to check if results are instances of certain type values and if so, apply the correct conversion functions. Apart from object-oriented programming languages where programs are sets of functions that define behavior, functions with the same name are not allowed and so if we wanted to have the same `toHtml` function to convert both `Integers` and `Doubles`, for example, we would have to give them different names, like `toHtmlInt` or `toHtmlDouble`. This restriction is quite annoying. Such an approach would like this (in *pseudo*-code):

```
1 if resultType == Int then
2    toHtmlInt(result);
3 elseif resultType == Double then
4    toHtmlDouble(result);
5 else (...)
```

As an alternative, other common conditional statements like `switch` (in Java), usually referred to as pattern matching could be used. It is basically the same as using *Ifs* but in a more elegant and concise way. Such an approach would like this (in *pseudo*-code):

```
1 switch(resultType) {
2    case Int: toHtmlInt(result);
3    case Double: toHtmlDouble(result);
4    (...)
5 }
```

Of course this approach implies that we need to obtain `resultType` as a textual, or in other format, representation of the result's type value. This seems feasible for simple types like `Int` but what if there are parameterized types, like `List[Int]` or `List[String]`? Or even `List[List[Int]]`? We start to reach the conclusion that such an approach is not flexible at all. We would have to exhaustively list *every* possible type value to be converted, which is not practical, and a very painful and error-prone mechanism.

In object-oriented languages, we could define Objects that encapsulate different conversions methods. We could have methods with the same name, like `toHtml`, but we would still have to inspect the result's type and explicitly call the appropriate method. Consider the following code as an example of the same approach, but in an object-oriented domain.

```
switch(resultType) {
   case Int: IntConversions.toHtml(result);
   case Double: DoubleConversions.toHtml(result);
   (...)
}
```

Well, we have surpassed naming the same conversion with different names but we're still with an ugly mechanism for type detection and conversion invocation. This is where Scala's implicits do their magic and provide an exceptionally flexible way of correctly invoking the appropriate conversion depending on the input type value. If you don't recall how Scala's implicits work you can read about it in section 4.4.

4.4.4.2 Conversions with Scala's Implicits

With Scala's implicits, because the compiler checks for implicit conversions available for converting the input type value, a lot of work is done in an abstract way and away from the programmers eyes. After correctly defining and declaring implicit conversions (as explained in section 4.3.3), we can simply invoke conversions functions like this (in Scala code):

```
val list = List(1,2,3,4)

3.toHtml()
(1 to 10).toHtml()
list.toHtml()
```

Because every value is an object in Scala, and because the compiler searches for the correct implicit conversion to apply, it is this simple to invoke conversions on identifiers without worrying about pre-detecting the identifier's type.

As explained in section 4.3.2.1, the conversion process of Visual Scala simply calls `toHtml` on the resulting identifier from evaluating expressions. This is highly flexible. If we want more conversions we simply need to define more methods and invoke them when necessary without having to expand a type-detecting mechanism presented in the first example, turning it into an unmanageable and huge control sequence that would need to be re-compiled or re-evaluated every time a new conversion is added. As such, Scala's implicits offer highly flexible ways of invoking methods according to specific types. Figure 4.7 shows an example of such flexibility in action.

Scala's implicit conversions are flexible when defining conversions for different data types. It is not possible for defining multiple conversions with the same name for the same type value. Although possible to define, When faced with ambiguity, the Scala compiler does not randomly invokes one of the conversion and instead throws an error message saying that conversions are ambiguous when invoking the methods. Figure 4.8 shows such behavior in action.

```
scala> implicit def fromList[A](list: List[A]) = new Object {
     | def toUnorderedList = <ul>{ list.map(e => <li>{ e }</li>) }</ul>
     | }
fromList: [A](list: List[A])java.lang.Object{def toUnorderedList: scala.xml.Elem}

scala> List(1,2,3).toUnorderedList
res0: scala.xml.Elem = <ul><li>1</li><li>2</li><li>3</li></ul>

scala> implicit def fromList2[A](list: List[A]) = new Object {
     | def toOrderedList = <ol>{ list.map(e => <li>{ e }</li>) }</ol>
     | }
fromList2: [A](list: List[A])java.lang.Object{def toOrderedList: scala.xml.Elem}

scala> List(1,2,3).toOrderedList
res1: scala.xml.Elem = <ol><li>1</li><li>2</li><li>3</li></ol>

scala> List(1,2,3).toUnorderedList
res2: scala.xml.Elem = <ul><li>1</li><li>2</li><li>3</li></ul>

scala> 2.toUnorderedList
<console>:10: error: value toUnorderedList is not a member of Int
              2.toUnorderedList
                ^

scala>
```

Figure 4.7: Example of the flexibility of Scala's implicits conversions

4.4.4.3 Conversions hierarchy

When dealing with parameterized type values, like `List` or `Map`, it is probably a best practice to define implicit conversions for the different sub-types for these data structures. It is possible to define general applicable conversions by parameterizing the input type values, as `List[A]` or `Map[A,B]`, however, sometimes the conversion we want to define does not apply to all possible sub-types. We may want to define an implicit conversion that makes sense for `List[Int]` but not for `List[String]`. Scala's implicits provide a flexible way of overcoming this by allowing to define parameterized conversions and more specific conversions and according to the input type value, Scala's compiler invokes the most specific method. Figure 4.9 shows such a mechanism in action. With this, the programmer can define multiple conversions for the same type value in different object, by controlling the exact conversions desired for the correct sub-types, without worrying with such a detection on general applicable conversions.

4.5 Implemented Conversions

In the previous sub-section we learned about the Conversions Library component of the Backend system and how it is structured. We saw a small example of a valid conversion source file for converting Scala's `Maps` to HTML tables and now we are going to present every conversion that was implemented as well as their results and difficulties.

Conversions are grouped and are to be presented according to the Scala's type value they convert.

4.5.1 `Iterable[A]`

Scala's `Iterables` are collection classes that provide method elements which return iterators over all the elements contained in the collection. The goal for converting `Iterables` was to

```
Welcome to Scala version 2.9.2 (OpenJDK Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> implicit def fromList1[A](x: List[A]) = new Object {
     | def toHtml = <ul>{ x.map(e => <li>{ e }</li>) }</ul>
     | }
fromList1: [A](x: List[A])java.lang.Object{def toHtml: scala.xml.Elem}

scala> implicit def fromList2[A](x: List[A]) = new Object {
     | def toHtml = <ol>{ x.map(e => <li>{ e }</li>) }</ol>
     | }
fromList2: [A](x: List[A])java.lang.Object{def toHtml: scala.xml.Elem}

scala> List("a","b").toHtml
<console>:10: error: type mismatch;
 found   : List[java.lang.String]
 required: ?{val toHtml: ?}
Note that implicit conversions are not applicable because they are ambiguous:
 both method fromList1 in object $iw of type [A](x: List[A])java.lang.Object{def toHtml: scala.xml.Elem}
 and method fromList2 in object $iw of type [A](x: List[A])java.lang.Object{def toHtml: scala.xml.Elem}
 are possible conversion functions from List[java.lang.String] to ?{val toHtml: ?}
              List("a","b").toHtml
                  ^

scala>
```

Figure 4.8: Scala's compiler dealing with ambiguity of implicits conversions

cover `List`, `Vector`, `Seq`, etc. In other words, to convert `Traversable` data structures.

4.5.1.1 toHtml

The default conversion `toHtml` converts Scala's `Iterables` to a paginated representation of lists. This is quite helpful because if the `Iterable` is rather big, the user can easily navigate through the elements without being overwhelmed by the high number of elements. Our graphical representation offers typical *Previous* and *Next* operations on the list and an alternated background color on each element for better visualizing them. The widget is also interactive, the user can drag elements and sort them as he wishes, producing a new `Iterable` with the new changes.

The visual representation is achieved with HTML, CSS and JavaScript, with no other external libraries being used. This conversion took a while to be completed mainly because of some styling issues. Figure 4.10 shows a screenshot of the paginated list obtained when evaluating `List(4,10,7,9,2,3,15,20,8,11,30,56,43,23,65,76)` and the next elements when iterating over the result.

4.5.1.2 toBarChart

Sometimes, lists or other variants, may represent some distribution of values that we want to see displayed as a bar chart. As a result, one of the first conversions we thought were quite useful was to convert `Iterables` to bar charts. Each bar of the chart has a different color, which is directly related to the value it represents, and has variable width, regarding the number of bars to display. Although it can be performed on `Iterables` containing other types than Numerals, this conversion is intended only for numeral types.

The bar chart is implemented through D3, which made it quite simple to achieve. Figure 4.11 shows a screenshot of `List(4,10,7,9,2,3,15,20,8,11,30,56,43,23,65,76)` being converted to a bar chart.

```
Welcome to Scala version 2.9.2 (OpenJDK Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> implicit def fromList[A](x: List[A]) = new Object {
     | def toHtml = <CONVERSION1> { x } </CONVERSION1>
     | }
fromList: [A](x: List[A])java.lang.Object{def toHtml: scala.xml.Elem}

scala> implicit def fromList1(x: List[Int]) = new Object {
     | def toHtml = <CONVERSION2> { x } </CONVERSION2>
     | }
fromList1: (x: List[Int])java.lang.Object{def toHtml: scala.xml.Elem}

scala> List("a","b").toHtml
res0: scala.xml.Elem = <CONVERSION1> ab </CONVERSION1>

scala> List(2,3,4).toHtml
res1: scala.xml.Elem = <CONVERSION2> 234 </CONVERSION2>

scala> List(2.5,3,4).toHtml
res2: scala.xml.Elem = <CONVERSION1> 2.53.04.0 </CONVERSION1>
```

Figure 4.9: Scala's compiler invoking more specific conversions rather than the parameterized one

4.5.1.3 TOBINARYTREE

Although usually represented by other data structures, binary trees can also be implemented as *arrays*, or lists. This special case usually refers to binary Heap trees, represented by lists for easily applying many algorithms. Programming beginners usually have a hard time visualizing binary trees and often draw them as they are using them in the code to correctly visualize the entire tree. The binary tree is constructed following the basic binary heap tree construct in which each element has their child nodes represented by the indexes *2i+1* and *2i+2*. This conversion assumes that the input list is a "valid" ordered heap tree, as it was not our intent to modify the input data to represent a binary tree.

This conversion was implemented using D3 and it the most time-consuming conversion because a slightly big learning curve was first to be traveled regarding coding with D3. Because it initially started with an animated binary tree being constructed, issues regarding showing the node's values took a long time to solve. Later, we dropped the animated tree and decided for a

Figure 4.10: Screenshot showing the default `toHtml` conversion on Scala's Iterables.

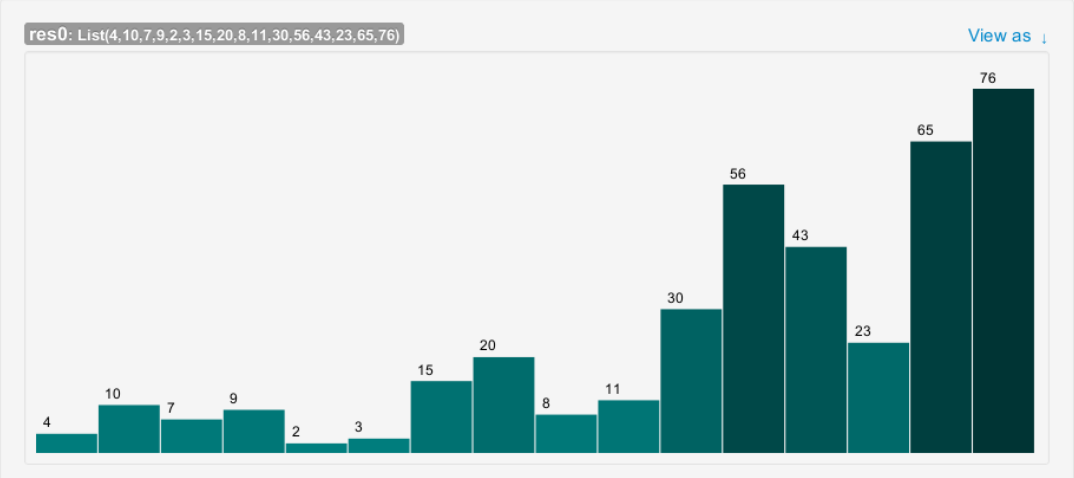

Figure 4.11: Screenshot showing the `toBarChart` conversion on Scala's Iterables.

static representation of the tree, this time with each node's value correctly presented. It was meant for the tree to be interactive but it was not yet done.

Figure 4.12 shows a screenshot of converting `Vector(2,3,4,7,8)` to a binary tree.

4.5.1.4 toHtmlList

Apart seeing `Iterables` as paginated lists, we also thought it would be nice to have an literal HTML list representation. This HTML list is constructed with the unordered list tag and it is also interactively sortable. The user can drag and sort elements, producing new `Iterables` when desirable.

This conversion was implemented using only HTML and JavaScript for dragging. Figure 4.13 shows a screenshot of converting `List(4,10,7,9,2,3,15,20,8,11,30,56,43,23,65,76)` to an unordered HTML list.

4.5.1.5 toOrderedList

Also, apart from seeing `Iterables` as paginated lists and unordered lists, it was only obvious to have a conversion for HTML ordered lists. This HTML list is constructed with the ordered list tag and it is also interactively sortable. The user can drag and sort elements, producing new `Iterables` when desirable.

This conversion was implemented using HTML and JavaScript for dragging. Figure 4.14 shows a screenshot of converting `List(4,10,7,9,2,3,15,20,8,11,30,56,43,23,65,76)` to an ordered HTML list.

4.5.2 `Map[A,B]`

Scala's `Maps` are Iterables consisting of pairs of keys and values (also named *mappings* or *associations*). `Maps` are very useful data structures to associate values to keys and to look them up and

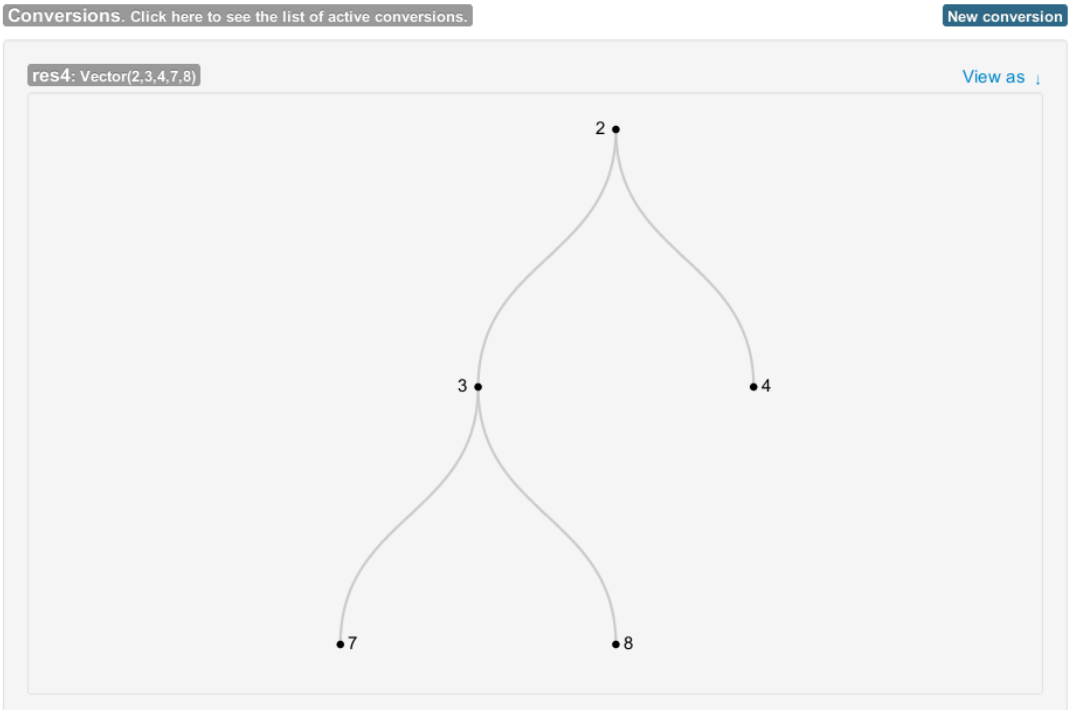

Figure 4.12: Screenshot showing the `toBinaryTree` conversion on Scala's Iterables.

manage them in very efficient ways.

4.5.2.1 toHtml

Since this data structure directly maps a value to a specific key our first thought was that the default conversion should be an HTML table. In this way, the user can visually map the values to the keys and see them spatially displayed in a table. We are confident this is the best visual representation for `Maps`.

This conversion was implemented using only HTML and the Bootstrap class *table*. Figure 4.15 shows a screenshot of converting `Map("Key 1" -> "Value 1", "Key 2" -> "Value 2", "Key 3" -> "Value 3", "Key 4" -> "Value 4")` to an HTML table.

4.5.2.2 toPieChart

Sometimes, this data structure represents mappings of entities to numerals (Integers, Doubles, etc.) and a useful way of visualization would be to render them as pie charts. Pie Charts were not applicable to `Iterables` because they lack to associate a key to the values contained in it. In this way, we can generate charts with associated labels to the values. Each element on the pie chart has a random generated color.

This conversion was achieved using D3's pie charts and took a considerable time to implement. Figure 4.16 shows a screenshot of converting `Map("Iterable" -> 5, "Map" -> 2, "String" -> 1, "AnyRef" -> 1)` to a D3 pie chart.

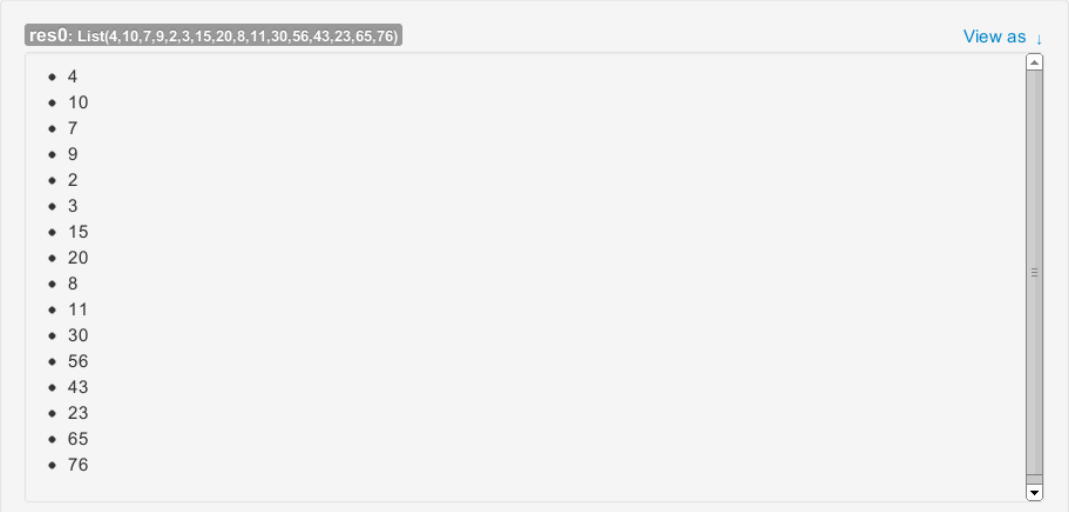

Figure 4.13: Screenshot showing the `toHtmlList` conversion on Scala's Iterables.

4.5.3 `String`

`Strings` are sequences of characters. All string literals in Java programs, and consequently in Scala (since it uses Java's Strings), such as "Visual Scala", are implemented as instances of the `String` class.

4.5.3.1 toHtml

We did not want for any visual representation of `Strings`. There is no apparent need to represent `Strings` as widgets, however, since they can contain special HTML characters like < or >, we needed that the default conversion addressed this issue. Consider the following `String`: "Hi, I'm a String. I have <some> special characters like < and >!". When trying to display that `String` on a web page, HTML would detect the special characters < and > and the contents between them would not be rendered. Also, if a `String` would contain HTML elements like "<h2> Howdy! </h2>", those elements would be rendered as valid HTML and produce unwanted results. And so, for `Strings` we had to convert HTML characters to their displayable representations, like `<` and `>`, respectively.

This conversion was implemented using only HTML. Figure 4.17 shows a screenshot of the default `toHtml` conversion on `Strings`.

4.6 Technologies

Libraries are sets of implementations of behavior, specific to one language, providing a well-defined application interface so the developer can invoke those behaviors. They provide an abstraction layer for the developer to invoke desired functionalities without having to implement them. Boosting productivity, the programmer can focus mainly on the design of the solution instead of re-occurring generic code for implementing some tasks.

Figure 4.14: Screenshot showing the `toOrderedList` conversion on Scala's Iterables.

A considerable number of technologies were used while implementing the dissertation' project, and are presented below.

4.6.1 Scala

4.6.1.1 Unfiltered

Unfiltered is a toolkit for servicing HTTP requests in Scala. It comes with a consistent vocabulary for handing requests on various server backends, without impeding direct access to their native interfaces. It allows for quickly adding web functionalities to a Scala application.

A giter8[6] *netty* template was applied to build the Unfiltered Scala Web server. With this template, we can manage receiving different HTTP requests with parameters, create and manipulate

[6]Command line tool to apply templates defined on GitHub, Nathan Hamblen at `https://github.com/n8han/giter8`

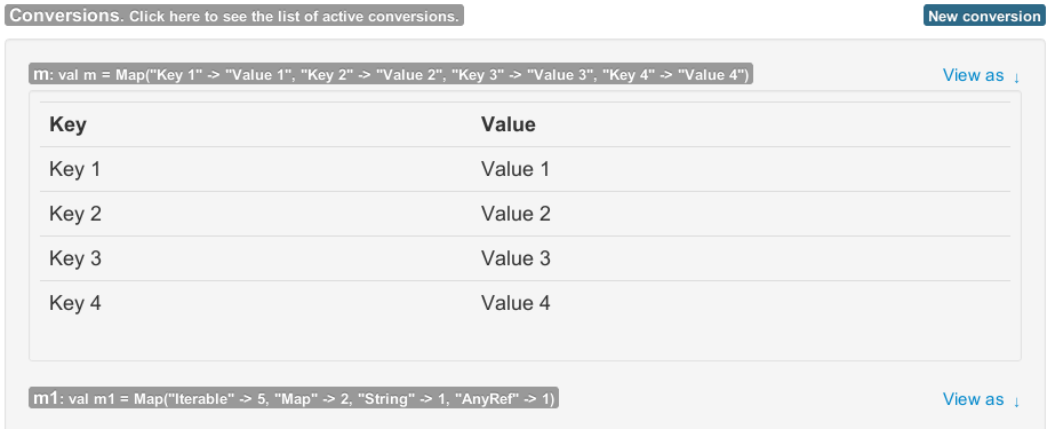

Figure 4.15: Screenshot showing the default `toHtml` conversion on Scala's Maps.

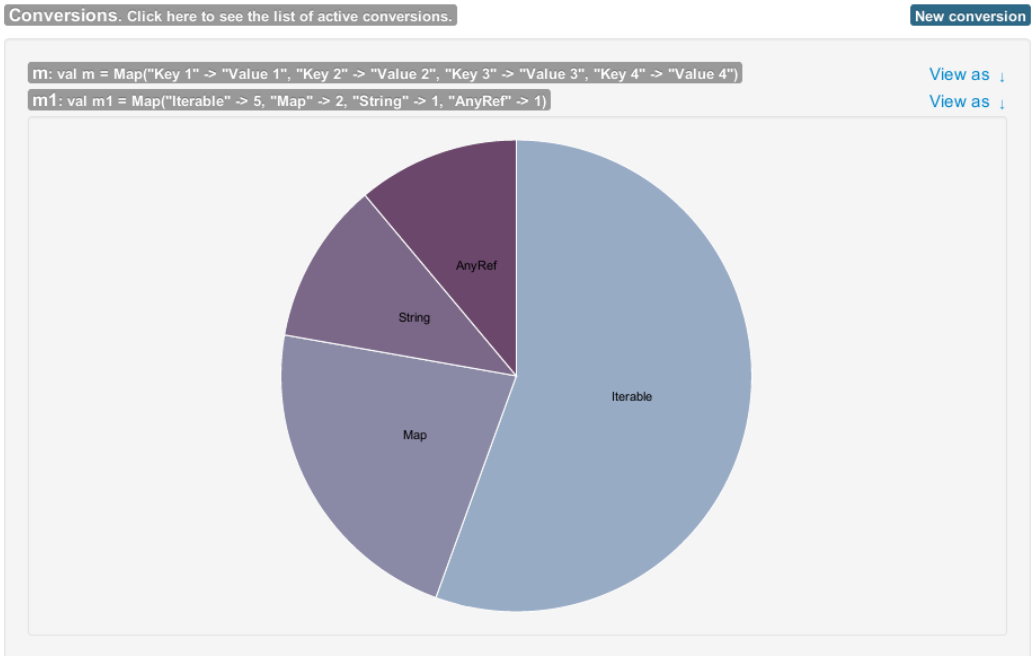

Figure 4.16: Screenshot showing the `toPieChart` conversion on Scala's Maps.

Cookies and deliver the results through HTTP.

4.6.1.2 ScalaInterpreterPane

ScalaInterpreterPane[7] is a Swing component for editing code in the Scala programming language and executing it in an interpreter. It offers an API to either instantiate a full GUI component containing a code editing section with the associated output results from the interpreter, as well as instantiating the separate parts.

ScalaInterpreterPane forks new Scala's Read-Eval-Print Loops with a top-level abstraction API for executing code and control results. This is helpful because we only needed a simple way of creating new instance of interpreters as well as an easy way to inspect results, regarding the obtained identifiers and result values.

As such, every instance of Scala's interpreters used in Visual Scala are created through ScalaInterpreterPane's API.

ScalaInterpreterPane was developed by Hanns Holger Rutz and released under the GNU Lesser General Public License.

[7] Free open-source project available at https://github.com/Sciss/ScalaInterpreterPane

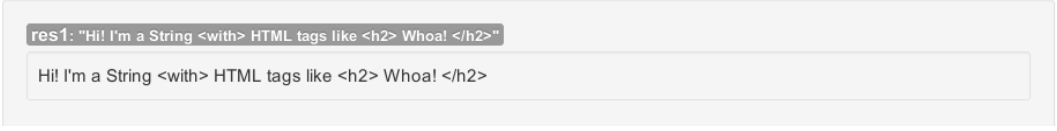

Figure 4.17: Screenshot showing the default `toHtml` conversion on Scala's Strings.

4.6.2 JavaScript, CSS and HTML

4.6.2.1 jQuery

jQuery is one of the most famous JavaScript libraries [Sur13], simplifying client-side scripting of HTML. It aims for a rapid development simplifying a wide range of operations like HTML document traversing, event handling, animations and AJAX interactions. Stating their website *jQuery is designed to change the way that you write JavaScript* [jQua]. Capabilities for creating plug-ins on top of the JavaScript library are also available and there are many jQuery plug-ins to choose from.

This library is used for creating abstractions for low-level interactions and animation. The standard operations of editing, creating and removing DOM elements are implement through jQuery quickly using the selector API implement according to the CSS selectors specification [jQub].

Maitaining and editing the DOM tree of the HTML page as well as all AJAX requests are done by jQuery. One plug-in is used, the jQuery Autosize plug-in to enable automatic height for *textarea* elements by Jack Moore.[8]

jQuery is free, open source software and licensed under the MIT License [jQu13].

4.6.2.2 Bootstrap

Twitter's Bootstrap is a free collection of software tools containing HTML and CSS-based design templates for typography, buttons, forms, and other interface components, providing rapid development of websites or web applications.

Advanced features like the grouping of buttons, buttons with drop-down lists, navigation, pagination, labels, advanced typographic capabilities thumbnails, formattable warning messages and progress bar comprise of a subset of the full potential of Bootstrap. It also contains JavaScript components that are based on the jQuery JavaScript library for additional user-interface elements such as dialogs, carousels, tooltips, drop-down, etc.

At its heart, Bootstrap *is just CSS, but it's built with Less, a flexible pre-processor that offers much more power and flexibility than regular CSS* [Dev11]. It is the most popular project in GitHub [Git13], compatible with all major browsers.

This library is used to style the main elements that are used for rendering the user-interface from the main interpreter output, labels, result boxes, collapsing result boxes and modal forms to the page title.

4.6.2.3 D3, Data Driven Documents

Data-Driven Documents, or D3, is a JavaScript library for displaying data as dynamic graphical representations. It is the successor of the previous Protovis[9] framework.

[8] You can read more about jQuery Autosize plug-in at `http://www.jacklmoore.com/autosize`
[9] You can read more at `http://mbostock.github.com/d3/tutorial/protovis.html`

The library, embedded within an HTML page, uses pre-built functions creating SVG elements, style them, add transitions or other dynamic effects. Standard CSS styling can also be applied to these SVG elements.

The data that is rendered can come in various formats, most commonly JSON, CSV, geoJSON, however, any format suits as long as the developer writes code that can convert it to supported ones.

This is the probably the most important library used in this dissertation. The abstraction level it provides allows for a more painless way to create graphical visualizations for HTML, allying with extensive documentation and examples. D3 also allows for a deeper control over the the final visual result [Via12]. D3's core contribution is a visualization "kernel" rather than a framework [BOH11].

The project's development environment needs to be easily extensible and we believe that with the popularity of D3 and the amount of tutorials and examples, users can quickly contribute with more Scala type values graphical conversions.

4.6.2.4 Session.js

Session.js is a JavaScript script that implements cookie-less JavaScript session variables. Used for storing and accessing variables that stay persistent throughout refreshes on the web page. Only when either the tab or browser is closed the session's variables are lost.

It has a very simple API comprising only four interactions. `Session.set(name, object)` stores a new object under the key `name` on the session. Consequently `Session.get(name)` returns the object stored under the provided key. The remaining operations are `Session.dump()` which returns the entire session variable with all it's values and `Session.clear()` which erases all contents.

The main purpose for using *Session.js* was to preserve previous results on the web page whenever the user refreshed it. Although all history is preserved server-side, if the user accidentally refreshed the web page, regarding client-side, all contents would be lost. And so, we keep track of the "instruction counter", every previous result object as well as the instructions history for navigating previous instructions submitted by the user.

4.7 Summary

In this chapter we described the implementation process of Visual Scala.

Visual Scala consists of a Client-Server system, in which the client side, or frontend, is a web application, implemented with standard web technologies: HTML, CSS and JavaScript. The server side, or backend, is written entirely in Scala implementing a web server responsible for managing HTTP requests as well as an evaluation module responsible for evaluating user code. The conversions library is part of the backend, however, it is a resource of the Scala evaluation module, as their sources are not compiled when building the system's backend.

A comprehensive overview of the technologies used is presented, regarding programming languages used, frameworks/libraries used and other open-source projects available for use.

Chapter 5

Conclusions

In this dissertation, we started by looking into the current state of the art regarding software development environments and related tools. This broad and more generalized research grew more and more specific regarding systems that provide data structures visualization and dynamic execution of code during runtime. From that research, we learned that several work has been done in data visualization systems. However, these systems do not offer dynamism when executing user code, lengthy re-compilations and restarts are needed. This static approach makes such systems "closed" during execution, one is able to develop code that, when compiled, is executed and produces graphical outputs. This is not what we wanted. Because those same systems were implemented using statically typed languages which generally struggle to provide that dynamic behavior, however, Visual Scala was implemented with a statically typed language, Scala, and provides the dynamism that we wanted. In our vision to walk towards live development environments, we needed a system that could be dynamically *injected*[1] with code to be executed during runtime.

Some examples were found that can be classified as live systems, detailed in section 2.2, where user code is evaluated as the programmer is typing it. Environments or tools which offered graphical representations of data structures were also found but none was found that could offer both. For example, Online Python Tutor (see section 2.2.5) provides great graphical visualizations on data structures throughout the execution trace of Python code, but does not provide a live environment for programming in Python, while still providing graphical representations of data. Julia (see section 2.2.4), provides a typical command-line interaction in which the programmer submits one expression at a time and see them evaluated and if applicable, a graphical visualization of the produced output. But while very powerful, specially regarding performance, Julia's goal is to be used for technical computing, in a more mathematical context and does not enable end-users to create new graphical conversions. From this research we felt confident that our vision for *Visual Scala* was truly innovative.

[1] As to say, submit code to be dynamically executed during runtime.

5.1 Summary of Hypotheses

This dissertation' fundamental research question can be described as:

> *Can Scala's Implicits provide a flexible and elegant way of implementing a conversions library for different Scala's native type values? What advantages or disadvantages arise from using such a mechanism? What lessons can we learn from it?*

From decomposing the main research topic other hypotheses and goals were obtained. The following list summarizes other hypotheses:

- **H1:** Can Scala's Implicits provide a flexible mechanism for converting data structures?
- **H2:** Can such a mechanism integrate a development environment for Scala?
- **H3:** Can such an environment provide a means to dynamically execute user code during *runtime*?
- **H4:** Can such an environment be easily extensible for new user-defined conversions? Even during *runtime*?

5.2 Main Results

From the previous list of hypotheses, the following items summarize the obtained results regarding each hypotheses and goal.

- **H1** *Can Scala's Implicits provide a flexible mechanism for converting data structures?*

  Scala's Implicits can, in fact, provide a flexible mechanism of converting Scala's type values into rich HTML representations. The lessons learned about this flexibility and results obtained are explained in more detail in section 4.4.4.

- **H2** *Can such a mechanism integrate a development environment for Scala?*

  Yes, the conversions mechanism represents the whole Conversions library implemented in Visual Scala. The results obtained from integrating such a library into an online development environment and the implemented conversions are presented in detail in section 4.5.

- **H3** *Can such an environment dynamically execute user code during runtime?*

  Yes, one of the goals was to develop an environment which could be seen as a *live* environment. By using Scala's standard interpreters for evaluating user code, Visual Scala can dynamically execute user code during *runtime*, in a *sandboxed* environment provided by the Java Virtual Machine.

- **H4** *Can such an environment be easily extensible for new user-defined conversions during runtime?*

  Yes, the implemented conversions library is part of Visual Scala's resources folder. As explained in section 4.3.3, the conversions library are a set of Scala source files containing the conversions methods for different data structures. The code from these source files is *injected* onto the target interpreter session for the end-user. Because of this approach the user can define new conversions through the use of a form, explained in section 4.2, and the code is *injected* onto the interpreter without the need for a re-compilation or restart. Changes and additions to the conversions library resource folder is also possible during *runtime* of the system because they are not part of the actual *running* code. We can safely conclude that Visual Scala is easily extensible.

5.3 Contributions

This dissertation research and development has provided with some contributions to the community.

State of the Art on LDEs This dissertation researched the current state of the art on *Live Development Environments*. They are not, clearly, a *new* idea, however they are still a working concept. LDEs do not possess an highly evolved maturity when comparing with IDEs, possibly because of the dynamic typing nature they have to tackle. Smalltalk, from 1996 is probably the best example of what a truly LDE would aim to be but no advances were made on bringing Smalltalk's features onto modern dynamic languages. Still, some good examples like Light Table or Adobe's Brackets might be a new future for LDEs.

Online LDE/REPL architecture for Scala We have contributed with a simple architecture for an online *Live Development Environment*/*Read-Eval-Print Loop* for the Scala programming language. Based on the standard Scala's REPL, Visual Scala results in an extensible architecture for a web application to simulate an approach of a LDE.

Implementing such an architecture From the architecture designed, this dissertation implemented that same architecture and successfully deployed Visual Scala to the Web. It contributed with a working web application that dynamically evaluates user code and converts data structures to rich graphical representations. Visual Scala can have different impacts and contributions for the community from helping beginners to better understand data structures and computation concepts to even serve as a collaborative tool for code development.

Conversion Library We have successfully developed an easily extensible and highly flexible *Conversions Library* that convert different Scala's native type values to rich, interactive HTML graphical representations. This library is easily extensible as it is not part of the running system's backend, located on the system's resources. It can serve as a case study for

the elegant flexibility Scala's Implicits provide when implementing mechanisms of invoking implicit conversions on data structures.

Open-Source code Visual Scala's code is open-source and made available for the community on GitHub. It is available at https://github.com/vascogrilo/LDE.

Article for the CDVE This dissertation overall work on Visual Scala resulted in a draft article for submission on the *10th International Conference on Cooperative Design, Visualization & Engineering*. We believe that with session sharing among different client's, Visual Scala can become a great tool for collaborative and cooperative development of algorithms and data structures as well as discussion or particular snippets of code.

5.4 Lessons Learned

Not everything in Software Engineering can be properly evaluated with the six-month time window that we were given to development of this dissertation. In order to correctly prepare and engage in controlled experiments with programmers using Visual Scala, it would take a least a year to gather meaningful results and opinions. As a result, we can only go so far as detailing the lessons learned from exploring this tool we have developed. Visual Scala, as expected, turned out to be an extensible live environment for the Scala programming language. From the implementation process of this web application we took some remarks, we learned that such a conversion component has many powerful advantages and benefits but also some disadvantages.

5.4.1 Advantages

One advantage we feel Visual Scala can offer is a better understanding environment for beginners in programming, or computer science. With the visual representations of data structures, and often interactive as well, we believe Visual Scala can help beginners in their understanding of computation concepts, specially regarding data structures. The author remembers his path when learning to program, and his first programming language experience was Haskell and if such a similar tool was available for Haskell, many concepts would have been quicker to grab and the experience would be richer. And so we believe Visual Scala has many advantages regarding to the learning curve and experience of programming and programming in Scala. Unfortunately, there was no sufficient time to conduct *quasi*-experiments with controlled audiences and because of this we cannot state with certainty that Visual Scala really helps productivity or understanding but we do believe it can. Specially in collaboration, with that feature on the list of future work, session sharing with other users can help collaborative development and teaching.

Apart from impacts it may have on the end-user, Visual Scala has other advantages.

Its conversion library is easily extensible. Since this library is part of the backend resources and not part of the running web server, one change to the conversions library doesn't require recompilation and execution of the web server. The contents of the library are simply loaded onto every new interpreter session and so, every changes made can take effect immediately. Even if

we're talking about extending the already loaded conversion in a specific session, it is quite simple because the user can do it freely during runtime, and the backend can do it as well since it only requires to submit code to be evaluated by the target interpreter. From this we can see that Visual Scala is highly extensible.

The environment supports dynamic evaluation of user code during runtime. This provides a key aspect for interaction, as the user continuously types expressions for evaluation and see them being interpreted and converted to "widgets". Because of this REPL-like approach, Visual Scala appeals for quicker and more interactive interactions, discarding some, sometimes, boring steps of compilation and execution triggering by the end-user. Specially when compared to an approach of having to edit a file-like code editor and trigger the evaluation of the entire source file. This would result in a static context of usage, where the user would make new changes and see the backend system re-evaluate previous expressions, or if he was to erase the contents and develop some new code he would lose all previous work. And so, the command-line approach seemed better for achieving the purpose of *liveness*.

5.4.2 Disadvantages

As a web application, with a backend system responsible for user code evaluation, Scala code that performs I/O operations are not recommended. Every `print` or `println` method will print whatever the user specifies to the backend system's log files. In order to provide to the user the results from these operations, the backend would have to read output files and send the contents to the frontend.

Not only from `prints` but every I/O operation like file manipulation, etc., is not the intended use for Visual Scala. These operations result in side-effects on server side and since the client's browser is on another separate system, the desired outcomes are not achieved.

Another disadvantage of the conversions library is that all conversions code must be *injected*, i.e., evaluated by the target Scala interpreter so it can be available to the users. As the conversions library grows in size, so does the time needed for the target interpreter to evaluate its code. This presents an ugly bottleneck on performance and speed of configuration of the interpreting session for end-users. Currently Visual Scala injects all conversions code onto the target interpreter which takes a few seconds before the main page can be presented to the user. However, this can be fixed by forking a new interpreter, presenting the main page to the user, and have a thread injecting in a parallel fashion, the conversions code. That way the user can get the main page in a quick way. Other approach would be for the user to select which conversion modules he wants active on the interpreter session, adding them one by one.

5.4.3 Flexibility of Scala's Implicits

Please refer to section 4.4.4 for an extensive discussion on the flexibility of Scala's implicit conversions regarding the implementation of the conversion mechanism.

5.5 Use Cases Scenarios

Despite the mail goal was to explore the flexibility of Scala's Implicits towards an Extensible Live Environment for Scala, this dissertation contributed with a successful project named Visual Scala, online and working. Visual Scala can have different applications/usages, or use cases scenarios.

For one, Visual Scala can act as a standard interpreter for Scala, as an online version. We believe it can also create a significant impact on the community with its flexible and extensible conversions library. By turning abstract data structures into rich, interactive HTML representations, a programmer can better understand, manipulate and reason about certain data structures and effects of algorithms. A beginner student in computation might learn in a quicker way when comparing to using other tools.

On another scenario, it can be used to generate graphical representations of data to be exported for including them in external sources. For example, generating bar charts of different data for later use in some report/article, etc.

Lastly, but not least, Visual Scala can act as an interesting and powerful collaborative platform for developing Scala code. Users might use it cooperatively to develop algorithms, teach other users about data structures, cooperate in implementing new graphical conversions and even discuss and reason about programming practices in Scala.

5.6 Future Work

Developing every possible graphical conversion for every Scala type value was not feasible. It would take a long time as it was simply not possible during the time we had for development of this dissertation. As a result, we aimed for extensibility and we hope that, when correctly deployed and advertised, the community would help enhancing Visual Scala with graphical conversions.

Although the time for this dissertation has ended, Visual Scala's project has not. We feel like it can be an important tool and that it can have an impact on the Scala community as well as in other programming communities.

We will keep on working on Visual Scala, developing more graphical conversions and some more features. Features like session sharing, where an user can evaluate some Scala code and share the current state of their session with other users using a specific URL. Other users would open up that URL and see the current state from the other developer and the environment would be shared among the two users, or more, eventually. This can help collaboration between Scala programmers.

General optimizations are also on the checklist for future work as we want Visual Scala to be as robust as it can be.

Also, since we feel it can have a large impact on the community, we will be working on turning Visual Scala into an online cooperative development environment for Scala. Through session sharing, we want for multiple users to share the same interpreting session and develop

Scala code, discuss code, collaboratively develop new conversions, and learn Scala with the help of others.

The author will be working with the supervisors and possibly with a team of developers who may want to take on Visual Scala from here on out.

Appendix A

Related Solutions Analysis

In chapter 2 different related solutions to our work were analysed regarding certain features and functionalities. For each artifact analysed we have present a table regarding the results from such an analysis. In this appendix we're aggregating all tables into one for an easier and quicker overview analysis of the different artifacts.

Table gathers all results from every table present on section 2.2.

Artifact	Dynamic code injection	Graphical data structures	Create new conversions	Intent
Scala Worksheet	*Yes*	*No*	*Possible because it is for Scala, but would not be rendered as graphics*	*Tool designed for simulating a live environment for Scala where code is evaluated while the user is typing*
Simply Scala	*Yes*	*No*	*Possible because it is for Scala, but would not be rendered as graphics*	*Web application designed to be an online version of the default Scala interpreter*
Scripster	*No*	*No*	*Possible because it is for Scala, but would not be rendered as graphics*	*Web application designed to be an online code editor for Scala, evaluating lines or chunks of code*
Julia Language	*Yes*	*Yes*	*Yes*	*Dynamic language for technical computing, offering an interpreter-like component for entering code and see it render graphically*
Online Python Tutor	*No*	*Yes*	*No*	*Powerful web application for visualizing the execution trace of Python code. Data structures have graphical visualizations and they change as we iterate through the code's execution*

Table A.1: Aggregated analysis of all artifacts from section 2.2

References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.

[Ber10] Rocky Bernstein. Debuggers in dynamic languages, June 2010.

[BKSE12] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[Bra08] Gilad Bracha. Dynamic ides for dynamic languages!, November 2008. http://gbracha.blogspot.pt/2008/11/dynamic-ides-for-dynamic-languages.html.

[Caz98] Walter Cazzola. Evaluation of object-oriented reflective models. In *In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th*, pages 3–540, 1998.

[CC86] Olivier Clarisse and Shi-Kuo Chang. Vicon: A visual icon manager. In *Visual Languages*, pages 151–190, New York, NY, USA, 1986. New York: Plenum Press.

[CDJ+97] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[Dev11] Twitter Developers. Bootstrap from twitter, 2011. https://dev.twitter.com/blog/bootstrap-twitter.

[Fow99] Martin Fowler. *Refactoring: Improving the Desing of Existing Code*. Addison-Wesley Professional, First edition, 1999.

[Git13] GitHub. Popular starred repositories, 2013. https://github.com/popular/starred.

[Guo13] Philip Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2013.

[Jav10] Bhavin Javia. Productive programmer: Using ide effectively & various small practices to improve productivity at xp days indore 2010, 2010. http://www.slideshare.net/bhavinjavia/productive-programmer-using-ide-effectively-and-various-small-practices-to-i Accessed Jan 2013.

[JF96] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2-3):181–202, 1996.

[jQua] jQuery. jquery - write less do more. Accessed January 2013 at http://jquery.com/.

[jQub] jQuery. jquery api - selectors. Accessed January 2013 at http://api.jquery.com/category/selectors/.

[jQu13] jQuery. License - jquery javascript library, 2013. http://docs.jquery.com/License.

[Kay03] Alan Kay. Dr. alan kay on the meaning of "object-oriented programming", 2003. http://www.purl.org/stefan_ram/pub/doc_kay_oop_en, Accessed Jan 2013.

[KG07] Suleyman Karabuk and F. Hank Grant. A common medium for programming operations-research models. *IEEE Softw.*, 24(5):39–47, 2007.

[KK11] Eugene Kindler and Ivan Krivý. Object-oriented simulation of systems with sophisticated control. *Int. J. General Systems*, 40(3):313–343, 2011.

[KS12] T. Kleenex and J.D. Suereth. *Scala in Depth*. Manning Pubs Co Series. O'Reilly Media, 2012.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Leu08] Ted Leung. Ide's and dynamic languages, July 2008. http://www.sauria.com/blog/2008/07/20/ides-and-dynamic-languages/.

[McC62] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[MM90] Ole Lehrmann Madsen and Boris Magnusson. Strong typing of object-oriented languages revisited. In *In OOPSLA-ECOOP '90 Proceedings, pages 140–150. ACM SIGPLAN Notices,25(10*, pages 140–150. ACM Press, 1990.

[Mye86] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 59–66, New York, NY, USA, 1986. ACM.

[Nor92] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition edition, 2008.

REFERENCES

[OVZS12] Michael C. Orsega, Bradley T. Vander Zanden, and Christopher H. Skinner. Experiments with algorithm visualization tool development. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 559–564. ACM, 2012.

[Pau07] L. D. Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[Pro13] Prototype. http://www.prototypejs.org/, January 2013.

[Scaa] Scala. The scala programming language. http://www.scala-lang.org/node/25, Accessed Jan 2013.

[Scab] Scala. A tour of scala: Unified types. http://www.scala-lang.org/node/128, Accessed Jan 2013.

[Smi77] D.C. Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Interdisciplinary systems research. Birkhuser, 1977.

[Smi82] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[Squ12] Squeak. Squeak smalltalk, 2012. http://www.squeak.org/Documentation/.

[Sur13] W3Techs Web Techonology Surveys. Usage of javascript libraries for websites, 2013. http://w3techs.com/technologies/overview/javascript_library/all.

[THP93] Walter F. Tichy, Nico Habermann, and Lutz Prechelt. Summary of the dagstuhl workshop on future directions in software engineering: February 17–21, 1992, schloß dagstuhl. *SIGSOFT Softw. Eng. Notes*, 18(1):35–48, January 1993.

[too13] The Dojo toolkit. http://dojotoolkit.org/, January 2013.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[Ung07] D. Ungar. Dynamic languages (in reactive environments) unleash creativity [programming languages]. *IEEE Software*, 24(5):72, 74, 2007.

[Via12] Datameer Cristophe Viau. What's behind our business infographics designer? d3.js of course, 2012. http://www.datameer.com/blog/uncategorized/whats-behind-our-business-infographics-designer-d3-js-of-course-2.html.

[Wic09] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009.

[Yeg08] Steve Yegge. 5. dynamic languages strike back (may 7, 2008), 2008.

[ZW98] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.